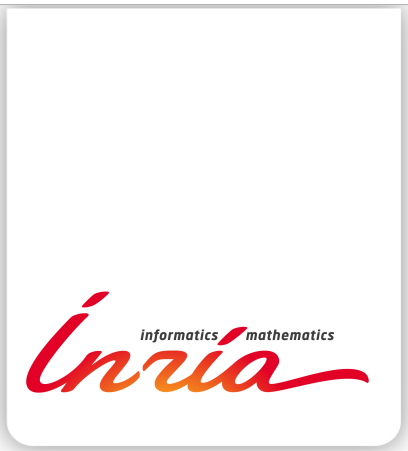
Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate

Karthikeyan Bhargavan, Bruno Blanchet, Nadim Kobeissi

RESEARCH REPORT
N° TBD
May 2017
Project-Team Prosecco

ISSN 0249-6399 ISRN INRIA/RR--TBD--FR+ENG



Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate

Karthikeyan Bhargavan, Bruno Blanchet, Nadim Kobeissi

Project-Team Prosecco

Research Report n° TBD — May 2017 — 51 pages

Abstract: TLS 1.3 is the next version of the Transport Layer Security (TLS) protocol. Its clean-slate design is a reaction both to the increasing demand for low-latency HTTPS connections and to a series of recent high-profile attacks on TLS. The hope is that a fresh protocol with modern cryptography will prevent legacy problems; the danger is that it will expose new kinds of attacks, or reintroduce old flaws that were fixed in previous versions of TLS. After 18 drafts, the protocol is nearing completion, and the working group has appealed to researchers to analyze the protocol before publication. This paper responds by presenting a comprehensive analysis of the TLS 1.3 Draft-18 protocol.

We seek to answer three questions that have not been fully addressed in previous work on TLS 1.3: (1) Does TLS 1.3 prevent well-known attacks on TLS 1.2, such as Logjam or the Triple Handshake, even if it is run in parallel with TLS 1.2? (2) Can we mechanically verify the computational security of TLS 1.3 under standard (strong) assumptions on its cryptographic primitives? (3) How can we extend the guarantees of the TLS 1.3 protocol to the details of its implementations?

To answer these questions, we propose a methodology for developing verified symbolic and computational models of TLS 1.3 hand-in-hand with a high-assurance reference implementation of the protocol. We present symbolic ProVerif models for various intermediate versions of TLS 1.3 and evaluate them against a rich class of attacks to reconstruct both known and previously unpublished vulnerabilities that influenced the current design of the protocol. We present a computational CryptoVerif model for TLS 1.3 Draft-18 and prove its security. We present RefTLS, an interoperable implementation of TLS 1.0-1.3 and automatically analyze its protocol core by extracting a ProVerif model from its typed JavaScript code.

Key-words: security protocols, verification, symbolic model, computational model, implementation, TLS

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Modèles vérifiés et implémentations de référence pour le candidat standard TLS 1.3

Résumé : TLS 1.3 est la prochaine version du protocole TLS (Transport Layer Security). Sa conception à partir de zéro est une réaction à la fois à la demande croissante de connexions HTTPS à faible latence et à une série d'attaques récentes de haut niveau sur TLS. L'espoir est qu'un nouveau protocole avec de la cryptographie moderne évitera d'hériter des problèmes des versions précédentes; le danger est que cela exposera à de nouveaux types d'attaques ou réintroduira d'anciens défauts qui ont été corrigés dans les versions précédentes de TLS. Après 18 versions préliminaires, le protocole est presque terminé, et le groupe de travail a appelé les chercheurs à analyser le protocole avant publication. Cet article répond en présentant une analyse globale du protocole TLS 1.3 Draft-18.

Nous cherchons à répondre à trois questions qui n'ont pas été entièrement traitées dans les travaux antérieurs sur TLS 1.3: (1) TLS 1.3 empêche-t-il les attaques connues sur TLS 1.2, comme Logjam ou Triple Handshake, même s'il est exécuté en parallèle avec TLS 1.2 ? (2) Peut-on vérifier mécaniquement la sécurité calculatoire de TLS 1.3 sous des hypothèses standard (fortes) sur ses primitives cryptographiques? (3) Comment pouvons-nous étendre les garanties du protocole TLS 1.3 aux détails de ses implémentations?

Pour répondre à ces questions, nous proposons une méthodologie pour développer des modèles symboliques et calculatoires vérifiés de TLS 1.3 en même temps qu'une implémentation de référence du protocole. Nous présentons des modèles symboliques dans ProVerif pour différentes versions intermédiaires de TLS 1.3 et nous les évaluons contre une riche classe d'attaques, pour reconstituer à la fois des vulnérabilités connues et des vulnérabilités précédemment non publiées qui ont influencé la conception actuelle du protocole. Nous présentons un modèle calculatoire dans CryptoVerif de TLS 1.3 Draft-18 et prouvons sa sécurité. Nous présentons RefTLS, une implémentation interopérable de TLS 1.0-1.3 et analysons automatiquement le coeur de son protocole en extrayant un modèle ProVerif à partir de son code JavaScript typé.

Mots-clés : protocoles cryptographiques, vérification, modèle symbolique, modèle calculatoire, implémentation, TLS

1 Introduction

The Transport Layer Security (TLS) protocol is widely used to establish secure channels on the Internet. It was first proposed under the name SSL [51] in 1994, and has undergone a series of revisions since, leading up to the standardization of TLS 1.2 [41] in 2008. Each version adds new features, deprecates obsolete constructions, and introduces countermeasures for weaknesses found in previous versions. The behavior of the protocol can be further customized via *extensions*, some of which are mandatory to prevent known attacks on the protocol.

One may expect that TLS clients and servers would use only the latest version of the protocol with all security-critical extensions enabled. In practice, however, many legacy variants of the protocol continue to be supported for backwards compatibility, and the everyday use of TLS depends crucially on clients and servers negotiating the most secure variant that they have in common. Securely composing and implementing the many different versions and features of TLS has proved to be surprisingly hard, leading to the continued discovery of high-profile vulnerabilities in the protocol.

A history of vulnerabilities. We identify four kinds of attacks that TLS has traditionally suffered from. *Downgrade* attacks enable a network adversary to fool a TLS client and server into using a weaker variant of the protocol than they would normally use with each other. In particular, version downgrade attacks were first demonstrated from SSL 3 to SSL 2 [80] and continue to be exploited in recent attacks like POODLE [66] and DROWN [8]. *Cryptographic* vulnerabilities rely on weaknesses in the protocol constructions used by TLS. Recent attacks have exploited key biases in RC4 [4, 79], padding oracles in MAC-then-Encrypt [5, 66], padding oracles in RSA PKCS#1 v1.5 [8], weak Diffie-Hellman groups [2], and weak hash functions [25]. *Protocol composition* flaws appear when multiple modes of the protocol interact in unexpected ways if enabled in parallel. For example, the renegotiation attack [74] exploits the sequential composition of two TLS handshakes, the Triple Handshake attack [17] composes three handshakes, and cross-protocol attacks [64, 80] use one kind of TLS handshake to attack another. *Implementation bugs* contribute to the fourth category of attacks on TLS, and are perhaps the hardest to avoid. They range from memory safety bugs like HeartBleed and coding errors like GotoFail to complex state machine flaws like SKIP and FREAK [15]. Such bugs can be exploited to bypass all the security guarantees of TLS, and their prevalence, even in widely-vetted code, indicates the challenges of implementing TLS securely.

Security proofs. Historically, when an attack is found on TLS, practitioners propose a temporary fix that is implemented in all mainstream TLS libraries, then a longer-term countermeasure is incorporated into a protocol extension or in the next version of the protocol. This has led to a attack-patch-attack cycle that does not provide much assurance in any single version of the protocol, let alone its implementations.

An attractive alternative would have been to develop security proofs that systematically demonstrated the absence of large classes of attacks in TLS. However, developing proofs for an existing standard that was not designed with security models in mind is exceedingly hard [70]. After years of effort, the cryptographic community only recently published proofs for the two main components of TLS: the *record* layer that implements authenticated encryption [63, 69], and the *handshake* layer that composes negotiation and key-exchange [52, 57]. These proofs required new security definitions and custom cryptographic assumptions, and even so, they apply only to abstract models of certain modes of the protocol. For example, the proofs do not account for low-level details of message formats, downgrade attacks, or composition flaws. Since such cryptographic proofs are typically carried out by hand, extending the proofs to cover all these details would require a prohibitive amount of work, and the resulting large proofs themselves

would need to be carefully checked.

A different approach taken by the protocol verification community is to *symbolically* analyze cryptographic protocols using simpler, stronger assumptions on the underlying cryptography, commonly referred to as the Dolev-Yao model [43]. Such methods are easy to automate and can tackle large protocols like TLS in all their gory detail, and even aspects of TLS implementations [35, 20]. Symbolic protocol analyzers are better at finding attacks, but since they treat cryptographic constructions as perfect black boxes, they provide weaker security guarantees than classic cryptographic proofs that account for probabilistic and computational attacks.

The most advanced example of mechanized verification for TLS is the ongoing miTLS project [23], which uses dependent types to prove both the symbolic and cryptographic security of a TLS implementation that supports TLS 1.0-1.2, multiple key exchanges and encryption modes, session resumption, and renegotiation. This effort has uncovered weaknesses in both the TLS 1.2 standard [17] and its other implementations [15], and the proof is currently being extended towards TLS 1.3.

Towards Verified Security for TLS 1.3. In 2014, the TLS working group at the IETF commenced work on TLS 1.3, with the goal of designing a faster protocol inspired by the success of Google's QUIC protocol [50]. Learning from the pitfalls of TLS 1.2, the working group invited the research community to contribute to the design of the protocol and help analyze its security even before the standard is published. A number of researchers, including the authors of this paper, responded by developing new security models and cryptographic proofs for various draft versions, and using their analyses to propose protocol changes. Cryptographic proofs were developed for Draft-5 [44], Draft-9 [58], and Draft-10 [61], which justified the core design of the protocol. A detailed symbolic model in Tamarin was developed for Draft-10 [39]. Other works studied specific aspects of TLS 1.3, such as key confirmation [46], client authentication [56], and downgrade resilience [16].

Some of these analyses also found attacks. The Tamarin analysis [39] uncovered a potential attack on the composition of pre-shared keys and certificate-based authentication, and this attack was prevented in Draft-11. A version downgrade attack was found in Draft-12 and its countermeasure in Draft-13 was proved secure [16]. A cross-protocol attack on RSA signatures was described in [53]. Even in this paper, we describe two vulnerabilities in 0-RTT client authentication that we discovered and reported, which influenced the subsequent designs of Draft-7 and -13.

After 18 drafts, TLS 1.3 is entering the final phase of standardization. Although many of its design decisions have now been vetted by multiple security analyses, several unanswered questions remain. First, the protocol has continued to evolve rapidly with every draft version, so many of the cryptographic proofs cited above are already obsolete and do not apply to Draft-18. Since many of these are manual proofs, it is not easy to update them and check all the proof steps. Second, none of these symbolic or cryptographic analyses, with the exception of [16], consider the composition of TLS 1.3 with legacy versions like TLS 1.2. Hence, they do not account for attacks like [53] that exploit weak legacy crypto in TLS 1.2 to break the modern cryptographic constructions of TLS 1.3. Third, none of these works addresses TLS 1.3 implementations. In this paper, we seek to cover these gaps with a new comprehensive analysis of TLS 1.3 Draft-18.

Our Contributions. We propose a methodology for developing mechanically verified models of TLS 1.3 alongside a high-assurance reference implementation of the protocol.

We present symbolic protocol models for TLS 1.3 written in ProVerif [31]. They incorporate a novel security model (described in §2) that accounts for all recent attacks on TLS, including those relying on weak cryptographic algorithms. In §3-5, we use ProVerif to evaluate various modes and drafts of TLS 1.3 culminating in the first symbolic analysis of Draft-18 and the first

composite analysis of TLS 1.3+1.2. Our analyses uncover known and new vulnerabilities that influenced the final design of Draft-18. Some of the features we study no longer appear in the protocol, but our analysis is still useful for posterity, to warn protocol designers and developers who may be tempted to reintroduce these problematic features in the future.

In §6, we develop the first machine-checked cryptographic proof for TLS 1.3 using the verification tool CryptoVerif [27]. Our proof reduces the security of TLS 1.3 Draft-18 to standard cryptographic assumptions over its primitives. In contrast to manual proofs, our CryptoVerif script can be more easily updated from draft-to-draft, and as the protocol evolves.

Our ProVerif and CryptoVerif models capture the protocol core of TLS 1.3, but they elide many implementation details such as the protocol API and state machine. To demonstrate that our security results apply to carefully-written implementations of TLS 1.3, we present RefTLS (§7), the first reference implementation of TLS 1.0-1.3 whose core protocol code has been formally analyzed for security. RefTLS is written in Flow, a statically typed variant of JavaScript, and is structured so that all its protocol code is isolated in a single module that can be automatically translated to ProVerif and symbolically analyzed against our rich threat model.

Our models and code are available at:

`https://github.com/inria-prosecco/reftls`

2 A Security Model for TLS

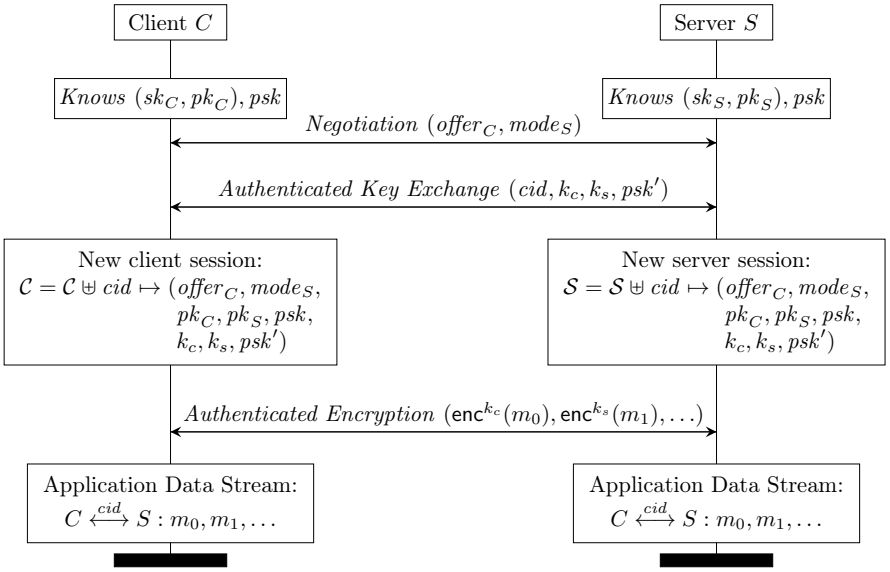


Figure 1: TLS Protocol Structure: Negotiation, then Authenticated Key Exchange (AKE), then Authenticated Encryption (AE) for application data streams. The server chooses $mode_S$, including the protocol version, the AKE mode, and the AE algorithm. In TLS 1.2, the AKE may use RSA, (EC)DHE, PSK, etc. and AE may use AES-CBC MAC-Encode-Encrypt, RC4, or AES-GCM. In TLS 1.3, the AKE may use (EC)-DHE, PSK, or PSK-(EC)DHE, and AE may use AES-GCM, AES-CCM, or ChaCha20-Poly1305. The use of one or more of (pk_C, pk_S, psk) in the session depends on the AKE mode.

Figure 1 depicts the progression of a typical TLS connection. Since a client and server may support different sets of features, they first *negotiate* a protocol mode that they have in common. In TLS, the client C makes an $offer_C$ and the server chooses its preferred $mode_S$, which includes the protocol version, the key exchange protocol, the authenticated encryption scheme, the Diffie-Hellman group (if applicable), and the signature and hash algorithms.

Then, C and S execute the negotiated *authenticated key exchange* protocol (e.g. Ephemeral Elliptic-Curve Diffie Hellman), which may use some combination of the long-term keys (e.g. public/private key pairs, symmetric pre-shared keys) known to the client and server. The key exchange ends by computing fresh symmetric keys (k_c, k_s) for a new session (with identifier cid) between C and S, and potentially a new pre-shared key (psk') that can be used to authenticate future connections between them.

In TLS, the negotiation and key exchange phases are together called the *handshake* protocol. Once the handshake is complete, C and S can start exchanging application data, protected by an authenticated encryption scheme (e.g. AES-GCM) with the session keys (k_c, k_s). The TLS protocol layer that handles authenticated encryption for application data is called the *record* protocol.

Security Goals for TLS. Each phase of a TLS connection has its own correctness and security goals. For example, during negotiation, the server must choose a $mode_S$ that is consistent with the client's $offer_C$; the key exchange must produce a secret session key, and so on. Although these intermediate security goals are important building blocks towards the security of the full TLS protocol, they are less meaningful to applications that typically use TLS via a TCP-socket-like API and are unaware of the protocol's internal structure. Consequently, we state the security goals of TLS from the viewpoint of the application, in terms of messages it sends and receives over a protocol session.

All goals are for messages between honest and authenticated clients and servers, that is, for those whose long-term keys (sk_C, sk_S, psk) are unknown to the attacker. If only the server is authenticated, then the goals are stated solely from the viewpoint of the client, since the server does not know whether it is talking to an honest client or the attacker.

Secrecy: If an application data message m is sent over a session cid between an honest client C and honest server S, then this message is kept confidential from an attacker who cannot break the cryptographic constructions used in the session cid.

Forward Secrecy: Secrecy (above) holds even if the long-term keys of the client and server (sk_C, pk_C, psk) are given to the adversary after the session cid has been completed and the session keys k_c, k_s are deleted by C and S.

Authentication: If an application data message m is received over a session cid from an honest and authenticated peer, then the peer must have sent the same application data m in a matching session (with the same parameters $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

Replay Prevention: Any application data m sent over a session cid may be accepted at most once by the peer.

Unique Channel Identifier: If a client session and a server session have the same identifier cid, then all other parameters in these sessions must match (same $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

These security goals encompass most of the standard security goals for secure channel protocols such as TLS. For example, secrecy for application data implicitly requires that the authenticated key exchange must generate secret keys. Authentication incorporates the requirement that

the client and server must have matching sessions, and in particular, that they agree on each others' identities as well as the inputs and outputs of negotiation. Hence, it prohibits client and server impersonation, and man-in-the-middle downgrade attacks.

The requirement for a unique channel identifier is a bit more unusual, but it allows multiple TLS sessions to be securely composed, for example via session resumption or renegotiation, without exposing them to credential forwarding attacks like Triple Handshake [17]. The channel identifier could itself be a session key or a value generated from it, but is more usually a public value that is derived from session data contributed by both the client and server [19].

Symbolic vs. Computational Models. Before we can model and verify TLS 1.3 against the security goals given above, we need to specify our protocol execution model. There are two different styles in which protocols have classically been modeled, and in this paper, we employ both of them. *Symbolic* models were developed by the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles is beyond the scope of this paper (see e.g. [30]); here we briefly outline their differences in terms of the two tools we will use.

ProVerif [28, 31] analyzes symbolic protocol models, whereas CryptoVerif [27] verifies computational models. The input languages of both tools are similar. For each protocol role (e.g. client or server) we write a *process* that can send and receive messages over public channels, trigger security events, and store messages in persistent databases.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels, and can manipulate them symbolically.

In CryptoVerif, messages are concrete bitstrings. Freshly generated nonces and keys are randomly sampled bitstrings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bitstrings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time CryptoVerif process running in parallel.

Authentication goals in both ProVerif and CryptoVerif are written as correspondences between events: for example, if the client triggers a certain event, then the server must have triggered a matching event in the past. Secrecy is treated differently in the two tools; in ProVerif, we typically ask whether the attacker can compute a secret, whereas in CryptoVerif, we ask whether it can distinguish a secret from a random bitstring.

The analysis techniques employed by the two tools are quite different. ProVerif searches for a protocol trace that violates the security goal, whereas CryptoVerif tries to construct a cryptographic proof that the protocol is equivalent (with high probability) to a trivially secure protocol. ProVerif is a push-button tool that may return that the security goal is true in the symbolic model, or that the goal is false with a counterexample, or that it is unable to conclude, or may fail to terminate. CryptoVerif is semi-automated, it can search for proofs but requires human guidance for non-trivial protocols.

We use both ProVerif and CryptoVerif for their complementary strengths. CryptoVerif can prove stronger security properties of the protocol under precise cryptographic assumptions, but the proofs require more work. ProVerif can quickly analyze large protocols to automatically find attacks, but a positive result does not immediately provide a cryptographic proof of security. Deriving sound cryptographic proofs using symbolic analysis is still an open problem for real-world protocols [38].

A Realistic Threat Model for TLS. We seek to analyze TLS 1.3 for the above security goals against a rich threat model that includes both classic protocol adversaries as well as new ones that apply specifically to multi-mode protocols like TLS. In particular, we model recent downgrade attacks on TLS by allowing the use of weak cryptographic algorithms in older versions of TLS. In our analyses, the attacker can use any of the following attack vectors to disrupt the protocol.

- **Network Adversary:** As usual, we assume that the attacker can intercept, modify, and send all messages sent on public network channels.
- **Compromised Principals:** The attacker can compromise any client or server principal P by asking for its long-term secrets, such as its private key (sk_P) or pre-shared key (psk). We do not restrict which principals can be compromised, but whenever such a compromise occurs, we mark it with a security event: `Compromised`($pk_{\mathtt{P}}$) or `CompromisedPSK`(psk). If the compromise event occurs after a session is complete, we issue a different security event: `PostSessionCompromise`(`cid`, $pk_{\mathtt{P}}$).
- **Weak Long-term Keys:** If the client or server has a weak key that the attacker may be able to break with sufficient computation, we treat such keys the same way as compromised keys and we issue a more general event:`WeakOrCompromised`($pk_{\mathtt{P}}$). This conservative model of weak keys is enough to uncover attacks like FREAK [15] that rely on the use of 512-bit RSA keys by TLS servers.
- **RSA Decryption Oracles:** TLS versions up to 1.2 use RSA PKCS#1 v1.5 encryption, which is known to be vulnerable to a form of padding oracle attack on decryption originally discovered by Bleichenbacher [32]. Although countermeasures to this attack have been incorporated into TLS, they remains hard to implement securely [65] resulting in continued attacks such as DROWN [8]. Furthermore, such padding oracles can sometimes even be converted to signature oracles for the corresponding private key [53].

  We assume that any TLS server (at any version) that enables RSA decryption may potentially be vulnerable to such attacks. We distinguish between two kinds of RSA key exchange: `RSA`(`StrongRSADecryption`) and `RSA`(`WeakRSADecryption`). In any session, if the server chooses the latter, we provide the attacker with a decryption and signature oracle for that private key.
- **Weak Diffie-Hellman Groups:** To account for attacks like Logjam [2], we allow servers to choose between strong and weak Diffie-Hellman groups (or elliptic curves), and mark the corresponding key exchange mode as `DHE`(`StrongDH`) or `DHE`(`WeakDH`). We conservatively assume that weak groups have size 1, so all Diffie-Hellman exponentiations in these groups return the same distinguished element `BadElement`.

  Even strong Diffie-Hellman groups typically have small subgroups that should be avoided. We model these subgroups by allowing a weak subgroup (of size 1) even within a strong group. A malicious client or server may choose `BadElement` as its public value, and then all exponentiations with this element as the base will also return `BadElement`. To avoid generating keys in this subgroup, clients and servers must validate the received public value.
- **Weak Hash Functions:** TLS uses hash functions for key derivation, HMAC, and for signatures. Versions up to TLS 1.2 use various combinations of MD5 and SHA-1, both of which are considered weak today, leading to exploitable attacks on TLS such as SLOTH [25].

  We model both strong and weak hash functions, and the client and server get to negotiate which function they will use in signatures. Strong hash functions are treated as one-way

functions in our symbolic model, whereas weak hash functions are treated as point functions that map all inputs to a constant value: `Collision`. Hence, in our model, it is trivial for the attacker to find collisions as well as second preimages for weak hash functions.

- **Weak Authenticated Encryption:** To model recent attacks on RC4 [4, 79] and TripleDES [24], we allow both weak and strong authenticated encryption schemes. For data encrypted with a weak scheme, irrespective of the key, we provide the adversary with a decryption oracle.

  A number of attacks on the TLS Record protocol stem from its use of a MAC-Encode-Encrypt construction for CBC-mode ciphersuites. This construction is known to be vulnerable to padding oracle attacks such as POODLE [66] and Lucky13 [5], and countermeasures have proved hard to implement correctly [3]. We model such attacks using a leaky decryption function. Whenever a client or server decrypts a message with this function, the function returns the right result but also leaks the plaintext to the adversary.

The series of threats described above comprise our conservative threat model for TLS 1.3, and incorporates entire classes of attacks that have been shown to be effective against older versions of the protocol, including Triple Handshake, POODLE, Lucky 13, RC4 NOMORE, FREAK, Logjam, SLOTH, DROWN. In most cases, we assume strictly stronger adversaries than have been demonstrated in practice, but since attacks only get better over time, our model seeks to be defensive against future attacks. It is worth noting that, even though TLS 1.3 does not itself support any weak ciphers, TLS 1.3 clients and servers will need to support legacy protocol versions for backwards compatibility. Our model enables a fine-grained analysis of vulnerabilities: we can ask whether TLS 1.3 connections between a client and a server are secure even if TLS 1.2 connections between them are broken.

Modeling the Threat Model in ProVerif. We encode our threat model as a generic ProVerif crypto library that can be used with any protocol. For each cryptographic primitive, our library contains constructors and destructors, that not only model the ideal behavior of strong cryptographic algorithms but also incorporates the possibility that honest protocol participants may support weak cryptographic algorithms.

Figure 2 displays our ProVerif model for Diffie-Hellman. We start by defining a type for groups; for simplicity, we only allow two groups (one strong, one weak). We then define a type for group elements and identify two distinguished elements that occur in every group, a basepoint `G`, and an element `BadElement` that belongs to a trivial subgroup. The function `dh_ideal` exponentiates a (public) element with a (secret) scalar; the equation describing its behavior encode the expected, ideal behaviour of Diffie-Hellman exponentiation. However, our protocol processes do not use `dh_ideal`; instead they call `dh_exp` which is parameterized by the group, and behaves differently depending on the strength of the group, and the validity of the element. If the group is strong and the element is a valid member of the group (that is, it does not belong to a small subgroup), then `dh_exp` behaves like `dh_ideal`. In all other cases, the result of `dh_exp` is the trivial element `BadElement`; that is, it is known to the attacker. The final `letfun` in the figure shows how Diffie-Hellman keys are generated. We note that our model of weak groups and bad elements is conservative and simplistic; we aim to verify the security of protocols that use strong groups, even if weak groups are catastrophically broken.

Similarly to our model of Diffie-Hellman, we write models for AEAD, hash functions, HMAC, RSA signatures and encryption. Using these primitives, all the cryptographic constructions of TLS 1.2 and 1.3 are built as derived functions. To understand our security theorems, it is important for the analyst to carefully inspect our cryptographic library and agree with its implicit and explicit assumptions.

```
type group.
const StrongDH: group [data].
const WeakDH: group [data].

type element.
fun e2b(element): bitstring [data].
const BadElement: element [data].
const G: element [data].

fun dh_ideal(element,bitstring):element.
equation forall x:bitstring, y:bitstring;
          dh_ideal(dh_ideal(G,x),y) =
          dh_ideal(dh_ideal(G,y),x).

fun dh_exp(group,element,bitstring):element
reduc forall g:group, e:element, x:bitstring;
       dh_exp(WeakDH,e,x) = BadElement
otherwise forall g:group, e:element, x:bitstring;
       dh_exp(StrongDH,BadElement,x) = BadElement
otherwise forall g:group, e:element, x:bitstring;
       dh_exp(StrongDH,e,x) = dh_ideal(e,x).

letfun dh_keygen(g:group) =
        new x:bitstring;
        let gx = dh_exp(g,G,x) in
        (x,gx).
```

Figure 2: A Model of Diffie-Hellman in ProVerif that allows the weak groups and allows elements in small subgroups.

Modeling and Verifying TLS 1.2 in ProVerif. To evaluate our threat model, and in preparation for our analysis of TLS 1.3, we symbolically analyze a model of TLS 1.2 using ProVerif. Our model includes TLS 1.2 clients and servers that support both RSA and Diffie-Hellman key exchanges, and are willing to use both weak and strong cryptography. We assume that clients are unauthenticated.

For illustration, Figure 3 shows the Diffie-Hellman handshake followed by the exchange of two application data fragments m_0, m_1. The handshake has four flights; the client sends a `ClientHello` offering a set of algorithms; the server responds with a `ServerHello` that chooses specific connection parameters, and then sends its certificate and (signed) key exchange message. The client responds with its own key exchange message and a `Finished` message with a MAC of the handshake using the connection master secret ms. The server completes the handshake by sending its own `Finished` message. These last two messages are meant to guarantee agreement on the keys and the handshake transcript, and they already encrypted with the new connection keys k. Once both have verified each others' finished messages, the application on top of TLS can start sending application data in the two directions.

We write ProVerif processes for TLS 1.2 clients and servers that exchange messages according to the protocol flow described above, and issue a sequence of events–`ClientOffers`, `ServerChooses`, `ClientFinished`, `ServerFinished`, `ClientSends`, `ServerReceives`–indicating their progress through the protocol. We then compose these processes with our threat model and add queries for message authenticity and secrecy.

For example, a secrecy query may ask whether the attacker can learn some application data message (say m_0) sent by the client over a particular connection (identified by the client and server random values and the server's public key):

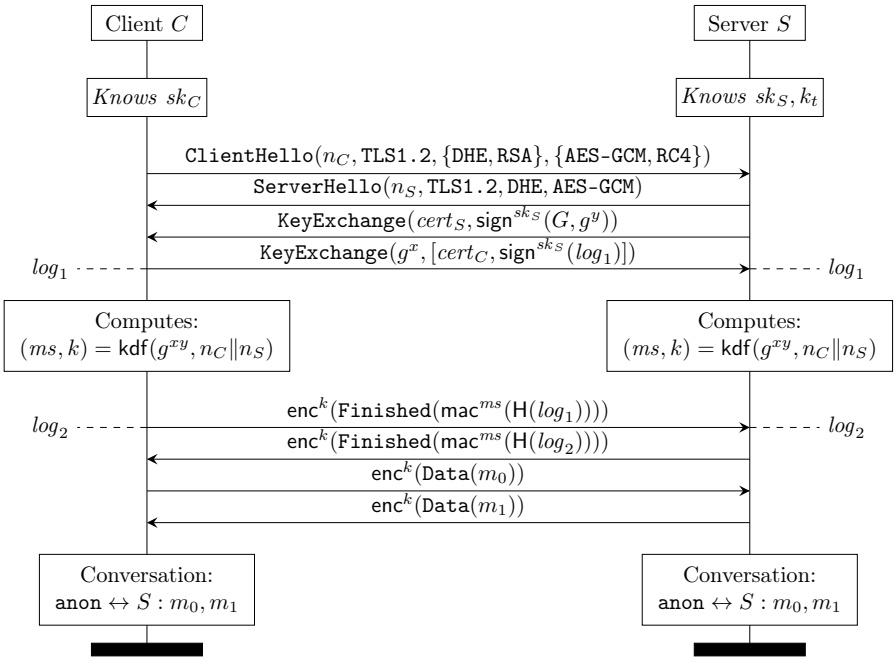


Figure 3: TLS 1.2 (EC)DHE handshake, no client authentication.

```
query cr:random, sr:random, p:pubkey;
      attacker(m_0(cr,sr,p)).
```

When we run ProVerif for this query, it finds a counter-example: the attacker can learn `m` if it can compromise server's private key (`WeakOrCompromised(pkS)`). To check whether this is the only case in which `m` is leaked, we refine the secrecy query and run ProVerif again. ProVerif again finds a counter-example: the attacker can learn `m` if the server chooses a weak Diffie-Hellman group (`ServerChoosesKex(DHE(WeakDH))`). In this way, we keep refining our queries until we obtain the strongest secrecy query that holds for our TLS 1.2 model, that is, until we stop finding attacks:

```
query cr:random, sr:random, p:pubkey, ms:bitstring, aek:ae_key,
      g:group, ra:rsa_alg,
      cb:bitstring, cr':random, sr':random, v:version;
      attacker(m_0(cr,sr,p)) ==>
      event(WeakOrCompromisedKey(p)) ||
      event(ServerChoosesAE(cr,sr,p,TLS12,WeakAE)) ||
      event(ServerChoosesKEX(cr,sr,p,TLS12,DHE(WeakDH))) ||
      event(ServerChoosesKEX(cr',sr',p,TLS12,RSA(WeakRSADecryption))) ||
      event(ServerChoosesHash(cr',sr',p,TLS12,WeakHash)) ||
      (event(PostSessionCompromisedKey(p)) &&
       event(ServerChoosesKEX(cr,sr,p,TLS12,RSA(ra)))) ||
```

This query is our main symbolic secrecy result for TLS 1.2; we similarly derive our strongest symbolic authenticaiton query. These two goals can be read as follows:

- **TLS 1.2 (Forward) Secrecy:** A message m sent by an honest client in a session *cid* to a server S cannot be known to the adversary unless one of the following conditions holds:

(1) the server's public key is weak or compromised, or
(2) the session uses weak authenticated encryption, or
(3) the session uses a weak Diffie-Hellman group, or
(4) the server uses weak RSA decryption with the same public key (in this or any other session), or
(5) the server uses a weak hash function for signing with the same public key (in any session), or
(6) the session uses an RSA key exchange and the server's public key was compromised after the session was complete.

- **TLS 1.2 Authenticity & Replay Protection:** Every message m accepted by an honest client in a session cid with some server S corresponds to a unique message sent by S on a matching session, unless one of the conditions (1)-(5) above holds.

Both these queries are verified by ProVerif in a few seconds. All the disjuncts (1)-(6) in these queries are necessary, removing any of them results in a counterexample discovered by ProVerif, corresponding to some well-known attack on badly configured TLS 1.2 connections.

Interestingly, the conditions (2), (3) are session specific, that is, only the sessions where these weak constructions are used are affected. In contrast, (4) and (5) indicate that the use of weak RSA decryption or a weak hash function in any session affects all other sessions that use the same server public key. As we shall see, this has an impact on the security of TLS 1.3 when it is composed with TLS 1.2. (6) encodes our forward secrecy goal, which does not hold for connections that use RSA key exchange, but holds for (EC)DHE.

We also verify our TLS 1.2 model for more advanced properties, such as unique channel identifiers; we find that using the connection key $cid = k_c$ does not yield a unique identifier. ProVerif finds a variant of the Triple Handshake attack, unless we implement the recommended countermeasure [71].

Verification Effort. The work of verifying TLS 1.2 can be divided into three tasks. We first modeled the threat model as a 400 line ProVerif library, but this library can now be reused for other protocols, including TLS 1.3. We then modeled the TLS 1.2 protocol in about 200 lines of ProVerif. Finally, we wrote about 50 lines of queries, both to validate our model (e.g. checking that the protocol completes in the absence of an attacker) and to prove our desired security goals. Most of the effort is in formalizing, refining, and discovering the right security queries. Although ProVerif is fully automated, verification gets more expensive as the protocol grows more complex. So, as we extend our models to cover multiple modes of TLS 1.3 composed with TLS 1.2, we sometimes need to simplify or restructure our models to aid verification.

3 TLS 1.3 1-RTT: Simpler, Faster Handshakes

In its simplest form, TLS 1.3 consists of a Diffie-Hellman handshake, typically using an elliptic curve, followed by application data encryption using an AEAD scheme like AES-GCM. The essential structure of 1-RTT has remained stable since early drafts of TLS 1.3. It departs from the TLS 1.2 handshake in two ways. First, the key exchange is executed alongside the negotiation protocol so the client can start sending application data along with its second flight of messages (after one round-trip, hence 1-RTT), unlike TLS 1.2 where the client had to wait for two message flights from the server. Second, TLS 1.3 eliminates a number of problematic features in TLS 1.2; it removes RSA key transport, weak encryption schemes (RC4, TripleDES, AES-CBC), and renegotiation; it requires group negotiation with strong standardized Diffie-Hellman groups, and it systematically binds session keys to the handshake log to prevent attacks like the

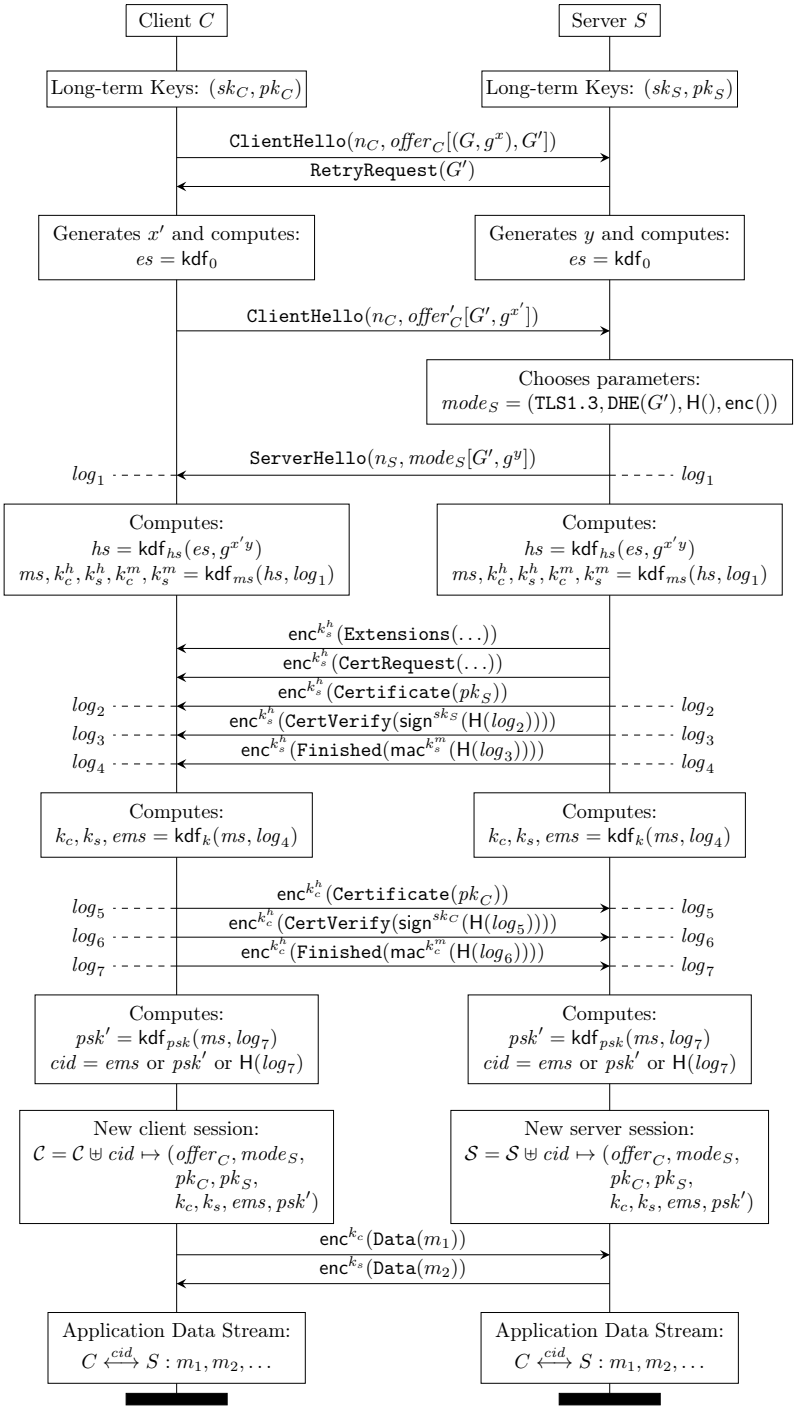


Figure 4: TLS 1.3 Draft-18 1-RTT Protocol. The protocol uses an (EC)DHE key exchange with server certificate authentication: client authentication and the `RetryRequest` negotiation steps are optional.

Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^k(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) = \mathsf{HMAC\text{-}H}^s(len_{\mathsf{H}()} \| \text{“}\texttt{TLS 1.3,}\text{”} \| l \| h \| \texttt{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where

$ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$

$hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1)$

$hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$

$k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \text{“”})$

$k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \text{“”})$

$k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \text{“”})$

$k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where

$ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4)$

$ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4)$

$ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$

$k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \text{“”})$

$k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where

$psk' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where

$es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk)$

$k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where

$ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$

$k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \text{“”})$

Figure 5: TLS 1.3 Draft-18 Key Schedule. The hash function H() is typically `SHA-256`, which has length $len_{\mathsf{H}()} = 32$ bytes.

Triple Handshake. In this section, we detail the protocol flow, we model it in ProVerif, and we analyze it alongside TLS 1.2 in the security model of §2.

1-RTT Protocol Flow. A typical 1-RTT connection in Draft 18 proceeds as shown in Figure 4. The first four messages form the negotiation phase. The client sends a `ClientHello` message containing a nonce n_C and an $offer_C$ that lists the versions, groups, hash functions, and authenticated encryption algorithms that it supports. For each group G that the client supports, it may include a Diffie-Hellman key share g^x. On receiving this message, the server chooses a $mode_S$ that fixes the version, group, and all other session parameters. Typically, the server chooses a group G for which the client already provided a public value, and so it can send its `ServerHello` containing a nonce n_S, $mode_S$ and g^y to the client. If none of the client's groups are acceptable, the server may ask the client (via `RetryRequest`) to resend the client hello with a key share $g^{x'}$ for the server's preferred group G'. (In this case, the handshake requires two round trips.)

Once the client receives the `ServerHello`, the negotiation is complete and both participants derive handshake encryption keys from $g^{x'y}$, one in each direction (k_c^h, k_s^h), with which they encrypt all subsequent handshake messages. The client and server also generate two MAC keys (k_c^m, k_s^m) for use in the `Finished` messages described below. The server then sends a flight of up to 5 encrypted messages: `Extensions` contains any protocol extensions that were not sent in the `ServerHello`; `CertRequest` contains an optional request for a client certificate; `Certificate` contains the server's X.509 public-key certificate; `CertVerify` contains a signature with server's private key sk_S over the log of the transcript so far (log_2); `Finished` contains a MAC with k_s^m over the current log (log_3). Then the server computes the 1-RTT traffic keys k_c, k_s and may immediately start using k_s to encrypt application data to the client.

Upon receiving the server's encrypted handshake flight, the client verifies the certificate, the signature, and the MAC, and if all verifications succeed, the client sends its own second flight consisting of an optional certificate `Certificate` and signature `CertVerify`, followed by a mandatory `Finished` with a MAC over the full handshake log. Then the client starts sending its own application data encrypted under k_c. Once the server receives the client's second flight, we consider the handshake complete and put all the session parameters into the local session databases at both client and server ($\mathcal{C}$, $\mathcal{S}$).

In addition to the traffic keys for the current session, the 1-RTT handshake generates two extra keys: *ems* is an exporter master secret that may be used by the application to bind authentication credentials to the TLS channel; psk' is a resumption master secret that may be used as a pre-shared key in future TLS connections between C and S.

The derivation of keys in the protocol follows a linear key schedule, as depicted on the right of Figure 4. The first version of this key schedule was inspired by OPTLS [58] and introduced into TLS 1.3 in Draft-7. The key idea in this design is to accumulate key material and handshake context into the derived keys using a series of HKDF invocations as the protocol progresses. For example, in connections that use pre-shared keys (see §5), the key schedule begins by deriving *es* from *psk*, but after the `ServerHello`, we add in $g^{x'y}$ to obtain the handshake secret *hs*. Whenever we extract encryption keys, we mix in the current handshake log, in order to avoid key synchronization attacks like the Triple Handshake.

Since its introduction in Draft-7, the key schedule has undergone many changes, with a significant round of simplifications in Draft-13. Since all previously published analyses of 1-RTT predate Draft-13, this leaves open the question whether the current Draft-18 1-RTT protocol is still secure.

Modeling 1-RTT in ProVerif. We write client and server processes in ProVerif that implement the message sequence and key schedule of Figure 4.

Our models are abstract with respect to the message formats, treating each message (e.g.

`ClientHello`$(\cdots)$) as a symbolic constructor, with message parsing modeled as a pattern-match with this constructor. This means that our analysis assumes that message serialization and parsing is correct; it won't find any attacks that rely on parsing ambiguities or bugs. This abstract treatment of protocol messages is typical of symbolic models; the same approach is taken by Tamarin [39]. In contrast, miTLS [23] includes a fully verified parser for TLS messages.

The key schedule is written as a sequence of ProVerif functions built using an HMAC function, `hmac`$(\mathtt{H}, \mathtt{m})$, which takes a hash function `H` as argument and is assumed to be a one-way function as long as $\mathtt{H} = \mathtt{StrongHash}$. All other cryptographic functions are modeled as described in §2, with both strong and weak variants.

Persistent state is encoded using tables. To model principals and their long-term keys, we use a global private table that maps principals (A) to their key pairs ((sk_A, pk_A)). To begin with, the adversary does not know any of the private keys in this table, but it can compromise any principal and obtain her private key. As described in §2, this compromise is recorded in ProVerif by an event `WeakOrCompromised`$(pk_{\mathtt{A}})$.

As the client and server proceed through the handshake they record security events indicating their progress. We treat the negotiation logic abstractly; the adversary gets to choose $offer_C$ and $mode_S$, and we record these choices as events (`ClientOffers`, `ServerChooses`) at the client and server. When the handshake is complete, the client and server issue events `ServerFinished`, `ClientFinished`, and store their newly established sessions in two private tables `clientSession` and `serverSession` (corresponding to $\mathcal{C}$ and $\mathcal{S}$). These tables are used by the record layer to retrieve the traffic keys k_c, k_s for authenticated encryption. Whenever the client or server sends or receives an application data message, it issues further events (`ClientSends`, `ServerReceives`, etc.) We use all these events along with the client and server session tables to state our security goals.

1-RTT Security Goals. We encode our security goals as ProVerif *queries* as follows:

- **Secrecy** for a message, such as m_1, is encoded using an auxiliary process that asks the adversary to guess the value of m_1; if the adversary succeeds, the process issues an event `MessageLeaked`$(cid, \mathtt{m_1})$. We then write a query to ask ProVerif whether this event is reachable.

- **Forward Secrecy** is encoded using the same query, but we explicitly leak the client and server's long-term keys (sk_C, sk_S) at the end of the session cid. ProVerif separately analyzes pre-compromise and post-compromise sessions as different *phases*; the forward secrecy query asks that messages sent in the first phase are kept secret even from attackers who learn the long-term keys in the second phase.

- **Authentication** for a message m_1 received by the server is written as a query that states that whenever the event `ServerReceives`$(cid, \mathtt{m_1})$ occurs, it must be preceded by three matching events: `ServerFinished`$(cid, \ldots)$, `ClientFinished`$(cid, \ldots)$, and `ClientSends`$(cid, \mathtt{m_1})$, which means that some honest client must have sent m_1 on a matching session. The authentication query for messages received by clients is similar.

- **Replay protection** is written as a stronger variant of the authentication query that requires *injectivity*: each `ServerReceives` event must correspond to a unique, matching, preceding `ClientSends` event.

- **Unique Channel Identifiers** are verified using another auxiliary process that looks up sessions from the `clientSession` and `serverSession` tables and checks that if the cid in both is the same, then all other parameters match. Otherwise it raises an event, and we ask ProVerif to prove that this event is not reachable.

In addition to the above queries, our scripts often include auxiliary queries about session keys and other session variables. We do not detail all our queries here; instead, we only summarize the main verification results. When we first ask ProVerif to verify these queries, it fails and provides counterexamples; for example, client message authentication does not hold if the client is compromised $\texttt{Compromised}(pk_{\texttt{C}})$ or unauthenticated in the session. We then refine the query by adding this failure condition as a disjunct, and run ProVerif again and repeat the process until the query is proved. Consequently, our final verification results are often stated as a long series of disjuncts listing the cases where the desired security goal does not hold.

Verifying 1-RTT in Isolation. For our model of Draft-18 1-RTT, ProVerif can prove the following secrecy query about all messages $(m_{0.5}, m_1, m_2)$:

- **1-RTT (Forward) Secrecy**: Messages m sent in a session between C and S are secret as long as the private keys of C and S are not revealed before the end of the session, and the server chooses a $mode_S$ with a strong Diffie-Hellman group, a strong hash function, and a strong authenticated encryption algorithm.

If we further assume that TLS 1.3 clients and servers only support strong algorithms, we can simplify the above query to show that all messages sent between uncompromised principals are kept secret. In the rest of this paper, we assume that TLS 1.3 only enables strong algorithms, but that earlier versions of the protocol may continue to support weak algorithms.

Messages m_1 from the client to the server enjoy strong authentication and protection from replays:

- **1-RTT Authentication (and Replay Prevention)**: If a message m is accepted by S over a session with an honest C, then this message corresponds to a unique message sent by the C over a matching session.

However the authentication guarantee for messages $m_{0.5}, m_1$ received by the client is weaker. Since the client does not know whether the server sent this data before or after receiving the client's second flight, the client and server sessions may disagree about the client's identity. Hence, for these messages, we can only verify a weaker property:

- **0.5-RTT Weak Authentication (and Replay Prevention)**: If a message m is accepted by C over a session with an honest S, then this message corresponds to a unique message sent by S over a server session that matches all values in the client session except (possibly) the client's public key pk_C, the resumption master secret psk', and the channel identifier cid.

We note that by allowing the server to send 0.5-RTT data, Draft-18 has weakened the authentication guarantees for all data received by an authenticated client. For example, if a client requests personal data from the server over a client-authenticated 1-RTT session, a network attacker could delay the client's second flight ($\texttt{Certificate} - \texttt{Finished}$) so that when the client receives the server's 0.5-RTT data, it thinks that it contains personal data, but the server actually sent data intended for an anonymous client.

Verifying TLS 1.3 1-RTT composed with TLS 1.2. We combine our model with the TLS 1.2 model described at the end of §2 so that each client and server supports both versions. We then ask the same queries as above, but only for sessions where the server chooses TLS 1.3 as the version in $mode_S$. Surprisingly, ProVerif finds two counterexamples.

First, if a server supports `WeakRSADecryption` with RSA key transport in TLS 1.2, then the attacker can use the RSA decryption oracle to forge TLS 1.3 server signatures and hence break our secrecy and authentication goals. This attack found by ProVerif directly corresponds to the cross-protocol Bleichenbacher attacks described in [53, 8]. It shows that removing RSA key

transport from TLS 1.3 is not enough, one must disable the use of TLS 1.2 RSA mode on any server whose certificate may be accepted by a TLS 1.3 client.

Second, if a client or server supports a weak hash function for signatures in TLS 1.2, then ProVerif shows how the attacker can exploit this weakness to forge TLS 1.3 signatures in our model, hence breaking our security goals. This attack corresponds to the SLOTH transcript collision attack on TLS 1.3 signatures described in [25]. To avoid this attack, TLS 1.3 implementations must disable weak hash functions in all supported versions, not just TLS 1.3.

After disabling these weak algorithms in TLS 1.2, we can indeed prove all our expected security goals about Draft-18 1-RTT, even when it is composed with TLS 1.2.

We may also ask whether TLS 1.3 clients and servers can be downgraded to TLS 1.2. If such a version downgrade takes place, we would end up with a TLS 1.2 session, so we need to state the query in terms of sessions where $mode_S$ contains TLS 1.2. ProVerif finds a version downgrade attack on a TLS 1.3 session, if the client and server support weak Diffie-Hellman groups in TLS 1.2. This attack closely mirrors the flaw described in [16]. Draft-13 introduced a countermeasure in response to this attack, and we verify that by adding it to the model, the downgrade attack disappears.

Although our models of TLS 1.3 and 1.2 are individually verified in a few seconds each, their composition takes several minutes to analyze. As we add more features and modes to the protocol, ProVerif takes longer and requires more memory. Our final composite model for all modes of TLS 1.3+1.2 takes hours on a powerful workstation.

4 0-RTT with Semi-Static Diffie-Hellman

In earlier versions of TLS, the client would have to wait for two round-trips of handshake messages before sending its request. 1-RTT in TLS 1.3 brings this down to one round trip, but protocols like QUIC use a "zero-round-trip" (0-RTT) mode, by relying on a *semi-static* (long-term) Diffie-Hellman key. This design was adapted for TLS in the OPTLS proposal [58] and incorporated in Draft-7 (along with a fix we proposed, as described below).

Protocol Flow. The protocol is depicted in Figure 6. Each server maintains a Diffie-Hellman key pair (s, g^s) and publishes a signed server configuration containing g^s. As usual, a client initiates a connection with a `ClientHello` containing its ephemeral key g^x. If a client has already obtained and cached the server's certificate and signed configuration (in a prior exchange for example), then the client computes a shared secret g^{xs} and uses it to derive an initial set of shared keys which can then immediately be used to send encrypted data. To authenticate its 0-RTT data, the client may optionally send a certificate and a signature over the client's first flight.

The server then responds with a `ServerHello` message that contains a fresh ephemeral public key g^y. Now, the client and server can continue with a regular 1-RTT handshake using the new shared secret g^{xy} in addition to g^{xs}.

The 0-RTT protocol continued to evolve from Draft-7 to Draft-12, but in Draft-13, it was removed in favor of a PSK-based 0-RTT mode. Even though Diffie-Hellman-based 0-RTT no longer exists in Draft-18, we analyze its security in this section, both for posterity and to warn protocol designers about the problems they should watch our for if they decide to reintroduce DH-based 0-RTT in a future version of TLS.

Verification with ProVerif. We modeled the protocol in ProVerif and wrote queries to check whether the 0-RTT data m_0 is (forward) secret and authentic. ProVerif is able to prove secrecy but finds that m_0 is not forward secret if the semi-static key s is compromised once the session is over. ProVerif also finds a Key Compromise Impersonation attack on authentication: if g^s is

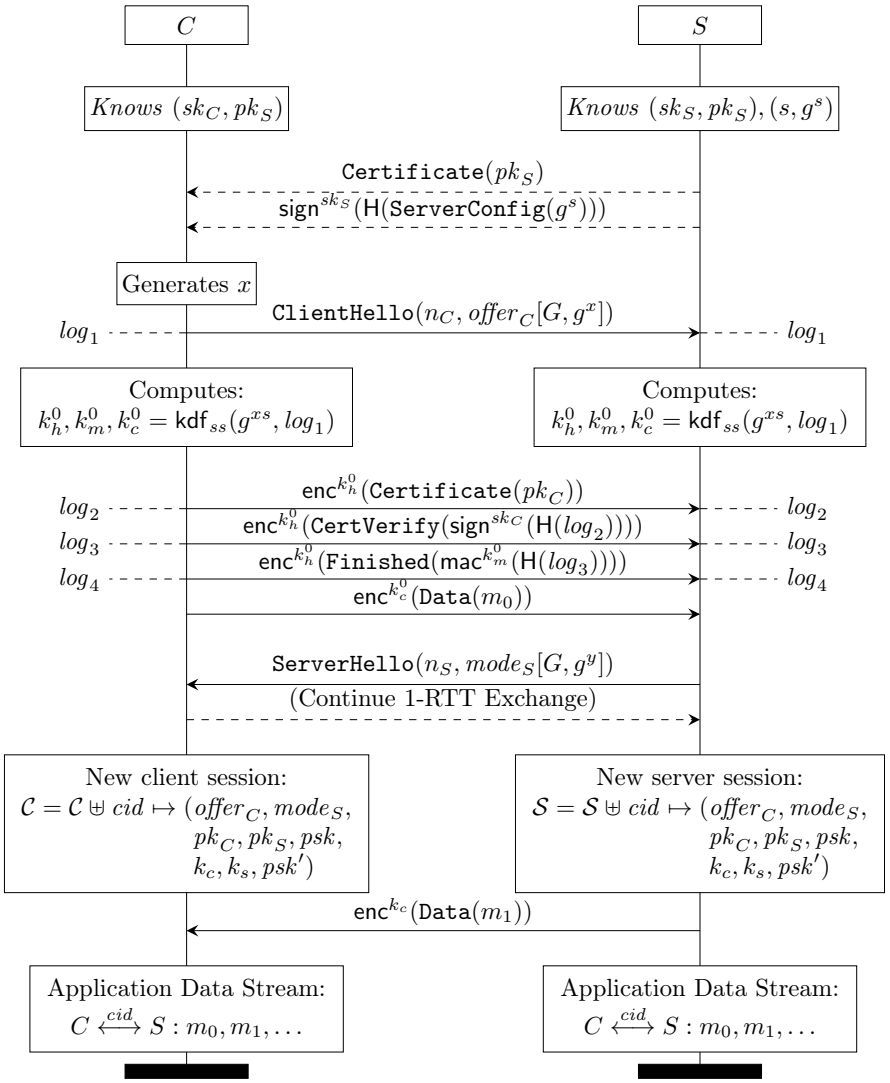


Figure 6: DH-based 0-RTT in TLS 1.3 Draft-12, inspired by QUIC and OPTLS.

compromised, then an attacker can forge 0-RTT messages from C to S. Furthermore, the 0-RTT flight can be replayed by an attacker and the server will process it multiple times, thinking that the client has initiated a new connection each time. In addition to these three concerns, which were documented in Draft-7, ProVerif also finds a new attack, explained below, that breaks 0-RTT authentication if the server's certificate is not included in the 0-RTT client signature.

Unknown Key Share Attack on DH-based 0-RTT in QUIC, OPTLS, and TLS 1.3. We observe that in the 0-RTT protocol, the client starts using g^s without having any proof that the server knows s. So a dishonest server M can claim to have the same semi-static key as S by signing g^s under its own key sk_M. Now, suppose a client connects to M and sends its client hello and 0-RTT data; M can simply forward this whole flight to S, which may accept it, because the semi-static keys match. This is an *unknown key share* (UKS) attack where C thinks it is talking to M but it is, in fact, connected to S.

In itself, the UKS attack is difficult to exploit, since M does not know g^{xs} and hence cannot decrypt or tamper with messages between C and S. However, if the client authenticates its 0-RTT flight with a certificate, then M can forward C's certificate (and C's signature) to S, resulting in a *credential forwarding* attack, which is much more serious. Suppose C is a browser that has a page open at website M; from this page M can trigger any authenticated 0-RTT HTTPS request m_0 to its own server, which then uses the credential forwarding attack to forward the request to S, who will process m_0 as if it came from C. For example, M may send a POST request that modifies C's account details at S.

The unknown key share attack described above applies to both QUIC and OPTLS, but remained undiscovered despite several security analyses of these protocols [45, 62, 58], because these works did not consider client authentication, and hence did not formulate an authentication goal that exposed the flaw. We informed the authors of QUIC and they acknowledged our attack. They now recommend that users who need client authentication should not use QUIC, and should instead move over to TLS 1.3. We also informed the authors of the TLS 1.3 standard, and on our suggestion, Draft-7 of TLS 1.3 included a countermeasure for this attack: the client signature and 0-RTT key derivation include not just the handshake log but also the cached server certificate. With this countermeasure in place, ProVerif proves authentication for 0-RTT data.

5 Pre-Shared Keys for Resumption and 0-RTT

Aside from the number of round-trips, the main cryptographic cost of a TLS handshake is the use of public-key algorithms for signatures and Diffie-Hellman, which are still significantly slower than symmetric encryption and MACs. So, once a session has already been established between a client and server, it is tempting to reuse the symmetric session key established in this session as a pre-shared symmetric key in new connections. This mechanism is called *session resumption* in TLS 1.2 and is widely used in HTTPS where a single browser typically has many parallel and sequential connections to the same website. In TLS 1.2, pre-shared keys (PSKs) are also used instead of certificates by resource-constrained devices that cannot afford public-key encryption. TLS 1.3 combines both these use-cases in a single PSK-based handshake mode that combines resumption, PSK-only handshakes, and 0-RTT.

Protocol Flow. Figure 7 shows how this mode extends the regular 1-RTT handshake; in our analysis, we only consider PSKs that are established within TLS handshakes, but similar arguments apply to PSKs that are shared out-of-band. We assume that the client and server have established a pre-shared key *psk* in some earlier session. The client has cached *psk*, but in order to remain state-less, the server has given the client a ticket containing *psk* encrypted under an encryption key k_t. As usual, the client sends a `ClientHello` with its ephemeral key

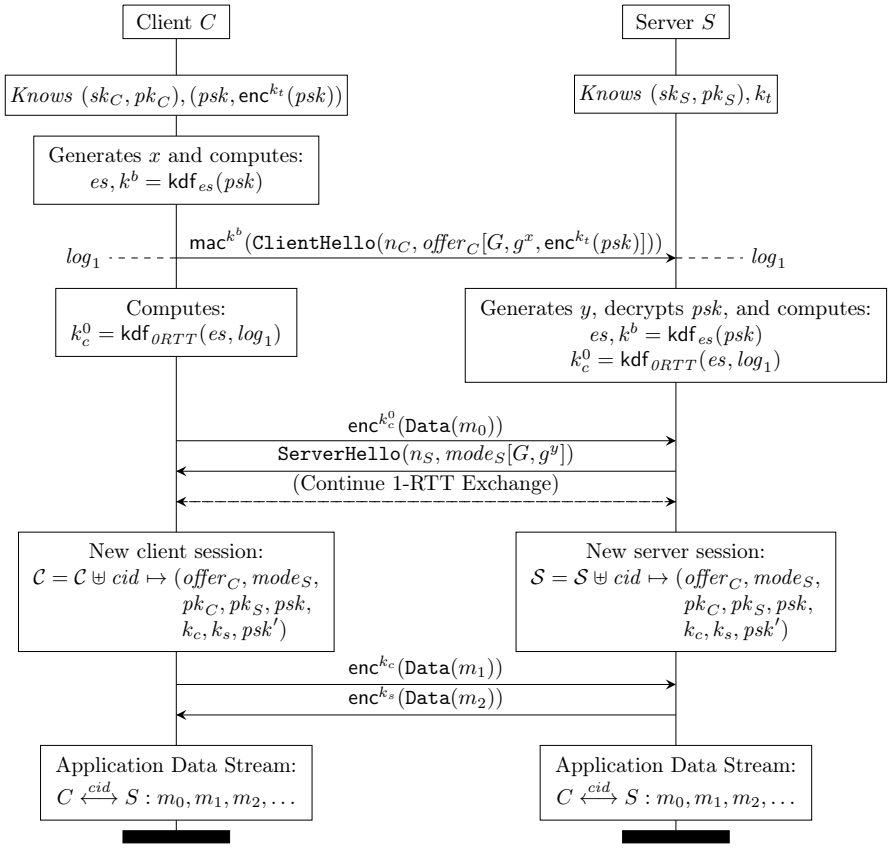


Figure 7: TLS 1.3 Draft-18 PSK-based Resumption and 0-RTT.

share g^x and indicates that it prefers to use the shared PSK *psk*. To prove its knowledge of *psk* and to avoid certain attacks (described below), it also MACs the `ClientHello` with a *binder* key k^b derived from the *psk*. The client can then use *psk* to already derive an encryption key for 0-RTT data m_0 and start sending data without waiting for the server's response. When the server receives the client's flight, it can choose to accept or reject the offered *psk*. Even if it accepts the *psk*, the server may choose to reject the 0-RTT data, it may choose to skip certificate-based authentication, and (if it does not care about forward secrecy) it may choose to skip the Diffie-Hellman exchange altogether. The recommended mode is PSK-DHE, where *psk* and g^{xy} are both mixed into the session keys. The server then sends back a `ServerHello` with its choice and the protocol proceeds with the appropriate 1-RTT handshake and completes the session.

Verifying PSK-based Resumption. We first model the PSK-DHE 1-RTT handshake (without certificate authentication) and verify that it still meets our usual security goals:

- **PSK-DHE 1-RTT (Forward) Secrecy** Any message m sent over a PSK-DHE session in 1-RTT is secret as long as the PSK *psk* and the ticket encryption key k_t are not compromised until the end of the session.
- **PSK-DHE 1-RTT Authentication and Replay Protection** Any message m received over a PSK-DHE session in 1-RTT corresponds to a unique message sent by a peer over a matching session (notably with the same *psk*) unless *psk* or k_t are compromised during the session.
- **PSK-DHE 1-RTT Unique Channel Identifier** The values psk', *ems*, and $\mathsf{H}(log_7)$ generated in a DHE or PSK-DHE session are all unique channel identifiers.

Notably, data sent over PSK-DHE is forward secret even if the server's long term ticket encryption key k_t is compromised after the session. In contrast, pure PSK handshakes do not provide this forward secrecy.

The authentication guarantee requires that the client and server must agree on the value of the PSK *psk*, and if this PSK was established in a prior session, then the unique channel identifier property says that the client and server must transitively agree on the prior session as well. An earlier analysis of Draft-10 in Tamarin [39] found a violation of the authentication goal because the 1-RTT client signature in Draft-10 did not include the server's `Finished` or any other value that was bound to the PSK. This flaw was fixed in Draft-11 and hence we are able to prove authentication for Draft-18.

Verifying PSK-based 0-RTT. We extend our model with the 0-RTT exchange and verify that m_0 is authentic and secret. The strongest queries that ProVerif can prove are the following:

- **PSK-based 0-RTT (Forward) Secrecy** A message m_0 sent from C to S in a 0-RTT flight is secret as long as *psk* and k_t are never compromised.
- **PSK-based 0-RTT Authentication** A message m_0 received by S from C in a 0-RTT flight corresponds to some message sent by C with a matching `ClientHello` and matching *psk*, unless the *psk* or k_t are compromised.

In other words, PSK-based 0-RTT data is not forward secret and is vulnerable to replay attacks. As can be expected, it provides a symmetric authentication property: since both C and S know the *psk*, if either of them is compromised, the attacker can forge 0-RTT messages.

An Attack on 0-RTT Client Authentication. Up to Draft-12, the client could authenticate its 0-RTT data with a client certificate in addition to the PSK. This served the following use

case: suppose a client and server establish an initial 1-RTT session (that outputs psk') where the client is unauthenticated. Some time later, the server asks the client to authenticate itself, and so they perform a PSK-DHE handshake (using psk') with client authentication. The use of psk' ensures continuity between the two sessions. In the new session, the client wants to start sending messages immediately, and so it would like to use client authentication in 0-RTT.

To be consistent with Draft-12, suppose we remove the outer binder MAC (using k^b) on the `ClientHello` in Figure 7, and we allow client authentication in 0-RTT. Then, if we model this protocol in ProVerif and ask the 0-RTT authentication query again, ProVerif finds a credential forwarding attack, explained next.

Suppose a client C shares psk with a malicious server M, and M shares a different psk' with an honest server S. If C sends an authenticated 0-RTT flight (certificate, signature, data m_0) to M, M can decrypt this flight using psk, re-encrypt it using psk', and forward the flight to S. S will accept the authenticated data m_0 from C as intended for itself, whereas C intended to send it only to M. In many HTTPS scenarios, as discussed in §4, M may be able to control the contents of this data, so this attack allows M to send arbitrary requests authenticated by C to S.

This attack was not discovered in previous analyses of TLS 1.3 since many of them did not consider client authentication; the prior Tamarin analysis [39] found a similar attack on 1-RTT client authentication but did not consider 0-RTT client authentication. The attacks described here and in [39] belong to a general class of *compound authentication* vulnerabilities that appear in protocols that compose multiple authentication credentials [19]. In this case, the composition of interest is between PSK and certificate-based authentication. We found a similar attack on 1-RTT server authentication in pure PSK handshakes.

In response to our attack, Draft-13 included a `resumption_context` value derived from the psk in the handshake hash, to ensure that the client's signature over the hash cannot be forwarded on another connection (with a different psk'). This countermeasure has since evolved to the MAC-based design showed in Figure 7, which has now been verified in this paper.

The Impact of Replay on 0-RTT and 0.5-RTT. It is now widely accepted that asynchronous messaging protocols like 0-RTT cannot be easily protected from replay, since the recipient has no chance to provide a random nonce that can ensure freshness. QUIC attempted to standardize a replay-prevention mechanism but it has since abandoned this mechanism, since it cannot prevent attackers from forcing the client to resend 0-RTT data over 1-RTT [72].

Instead of preventing replays, TLS 1.3 Draft-18 advises applications that they should only send non-forward-secret and idempotent data over 0-RTT. This recommendation is hard to systematically enforce in flexible protocols like HTTPS, where all requests have secret cookies attached, and even GET requests routinely change state.

We argue that replays offer an important attack vector for 0-RTT and 0.5-RTT data. If the client authenticates its 0-RTT flight, then an attacker can replay the entire flight to mount *authenticated replay* attacks. Suppose the (client-authenticated) 0-RTT data asks the server to send a client's bank statement, and the server sends this data in a 0.5-RTT response. An attacker who observes the 0-RTT request once, can replay it any number of times to the server from anywhere in the world and the server will send it the user's (encrypted) bank statement. Although the attacker cannot complete the 1-RTT handshake or read this 0.5-RTT response, it may be able to learn a lot from this exchange, such as the length of the bank statement, and whether the client is logged in.

In response to these concerns, client authentication has now been removed from 0-RTT. However, we note that similar replay attacks apply to 0-RTT data that contains an authentication cookie or OAuth token. We highly recommend that TLS 1.3 servers should implement a replay cache (based on the client nonce n_C and the ticket age) to detect and reject replayed 0-RTT

data. This is less practical in server farms, where time-based replay mitigation may be the only alternative.

6 Computational Analysis of TLS 1.3 Draft-18

Our ProVerif analysis of TLS 1.3 Draft-18 identifies the necessary conditions under which the symbolic security guarantees of the protocol hold. We now use the tool CryptoVerif [27] to see whether these conditions are sufficient to obtain cryptographic security proofs for the protocol in a more precise computational model. In particular, under the assumption that the algorithms used in TLS 1.3 Draft-18 satisfy certain strong cryptographic assumptions, we prove that the protocol meets our security goals.

Proofs in the computational model are hard to mechanize, and CryptoVerif offers less flexibility and automation than ProVerif. To obtain manageable proofs, we focus only on TLS 1.3 (we do not consider TLS 1.2) and we ignore downgrade attacks. Moreover, we proceed modularly: we first prove some lemmas on primitives, and we split the protocol into three pieces and prove them separately using CryptoVerif, before composing them manually to obtain a proof for the full protocol.

6.1 A Short Reminder on CryptoVerif

Processes, contexts, adversaries. CryptoVerif mechanizes proofs by sequences of games, similar to those written on paper by cryptographers [76, 13]. It represents protocols and cryptographic games in a probabilistic process calculus. We refer the reader to [27] for details on this process calculus. We will explain the necessary constructs as they appear.

Even though CryptoVerif can evaluate the probability of success of an attack as a function of the number of sessions and the probability of breaking each primitive (exact security), for simplicity, we consider here the asymptotic framework in which we only show that the probability of success of an attack is negligible as a function of the security parameter η. (A function f is *negligible* when for all polynomials q, there exists $\eta_o \in \mathbb{N}$ such that for all $\eta > \eta_0$, $f(\eta) \leq \frac{1}{q(\eta)}$.) All processes run in polynomial time in the security parameter and manipulate bitstrings of polynomially bounded length.

A *context* C is a process with one or several holes $[\,]$. We write $C[P_1, \ldots, P_n]$ for the process obtained by replacing the holes of C with $P_1, \ldots, P_n$ respectively. An *evaluation context* is a context with one hole, generated by the following grammar:

$C ::=$	evaluation context
$[\,]$	hole
newChannel $c; C$	channel restriction
$Q \mid C$	parallel composition
$C \mid Q$	parallel composition

The channel restriction **newChannel** $c; Q$ restricts the channel name c, so that communications on this channel can occur only inside Q, and cannot be received outside Q or sent from outside Q. The parallel composition $Q_1 \mid Q_2$ makes simultaneously available the processes defined in Q_1 and Q_2. We use evaluation contexts to represent *adversaries*.

Indistinguishability. A process can execute events, by two constructs: **event** $e(M_1, \ldots, M_n)$ executes event e with arguments $M_1, \ldots, M_n$, and **event_abort** e executes event e without argument and aborts the game. After finishing execution of a process, the system produces two results: the sequence of executed events $\mathcal{E}$, and the information whether the game aborted ($a =$

abort, that is, executed **event_abort**) or terminated normally ($a = 0$). These events and result can be used to distinguish games, so we introduce an additional algorithm, a *distinguisher* D that takes as input the sequence of events $\mathcal{E}$ and the result a, and returns true or false. Distinguishers must run in time polynomial in the security parameter. We write $\Pr[Q \rightsquigarrow_\eta D]$ for the probability that the process Q executes events $\mathcal{E}$ and returns a result a such that $D(\mathcal{E}, a) = \mathsf{true}$, with security parameter η.

Definition 1 (Indistinguishability)**.** *We write $Q \approx^V Q'$ when, for all evaluation contexts C acceptable for Q and Q' with public variables V and all distinguishers D, $|\Pr[C[Q] \rightsquigarrow_\eta D] - \Pr[C[Q'] \rightsquigarrow_\eta D]|$ is a negligible function η.*

Intuitively, $Q \approx^V Q'$ means that an adversary has a negligible probability of distinguishing Q from Q', when it has access to the variables in the set V. When V is empty, we omit it.

The condition that C is acceptable for Q and Q' with public variables V is a technical condition that guarantees that $C[Q]$ and $C[Q']$ are well-formed. The public variables V are the variables defined in Q and Q' that C is allowed to access directly.

The relation $\approx^V Q'$ is an equivalence relation, and $Q \approx^V Q'$ implies $C[Q] \approx^{V'} C[Q']$ for all evaluation contexts C acceptable for Q and Q' with public variables V and all $V' \subseteq V \cup \mathsf{var}(C)$, where $\mathsf{var}(C)$ is the set of variables of C.

Security assumptions on cryptographic primitives are given to CryptoVerif as indistinguishability properties that it considers as axioms.

Secrecy. Intuitively, secrecy means that the adversary cannot distinguish between the secrets and independent random values. This definition corresponds to the "real-or-random" definition of security [1]. A formal definition in CryptoVerif can be found in [29]. In this paper, we use the characterization given in Lemma 1 below. Let us first explain some CryptoVerif constructs used in this lemma. The replication $!^{i \leq n} Q$ represents n copies of the process Q in parallel, indexed by $i \in [1, n]$, where n is polynomial in the security parameter η. In CryptoVerif, all variables defined under a replication are implicitly arrays indexed by the replication indices: if Q defines a variable x under $!^{i \leq n}$, the value of x is in fact stored in $x[i]$. The definition of x is executed at most once for each i, so that all values of x are stored in distinct array cells. The **find** construct allows one to read these array cells: **find** $u = i' \leq n$ **suchthat defined**$(x_1[i'], \ldots, x_m[i']) \wedge M$ **then** P **else** P' looks for an index $i' \in [1, n]$ such that $x_1[i'], \ldots, x_m[i']$ are defined and M is true. When such an index is found, it is stored in u, and process P is executed. Otherwise, process P' is executed. The term M may refer to $x_1[i'], \ldots, x_m[i']$ and the process P may refer to $x_1[u], \ldots, x_m[u]$ since these variables are guaranteed to be defined. We sometimes use the same name for u and i'; in this case, we simply write **find** $u \leq n$ **suchthat** $\ldots$. The input $c[i](x : T); P$ receives a message on a message on channel $c[i]$ and stores it in x if it is in the set of bitstrings T (T stands for "type"), and executes P. If the received message is not in T, the process blocks. The channel $c[i]$ consists of a channel name c and indices, here i. Very often, the indices of channels correspond to the indices of replications above the input: that allows the sender to tell precisely to which copy of the process the message should be sent. Similarly, the output $\overline{c[i]}\langle M \rangle; Q$ sends message M on channel $c[i]$. After the output, the control is passed to the receiver process, which continues execution. The process Q that follows the output consists of inputs, possibly under replications and parallel compositions; these inputs will be executed when a message is sent to them. Finally, the restriction **new** $y : T; P$ chooses uniformly a random element of T, stores it in y and executes P.

Lemma 1. *Let Q be a process that does not contain* **event_abort**. *Let*

$$R_x^0 = !^{i_s \leq n_s}\, c_s[i_s](u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \textbf{if defined}(x[u_1, \ldots, u_m])\ \textbf{then}\ \overline{c_s[i_s]}\langle x[u_1, \ldots, u_m] \rangle$$

$$R_x^1 = !^{i_s \leq n_s}\, c_s[i_s](u_1 : [1, n_1], \ldots, u_m : [1, n_m]); \textbf{if } \textbf{defined}(x[u_1, \ldots, u_m]) \textbf{ then}$$
$$\quad \textbf{find } u_{s1} = i_{s1} \leq n_s \textbf{ suchthat } \textbf{defined}(y[i_{s1}], u_1[i_{s1}], \ldots, u_m[i_{s1}]) \wedge$$
$$\quad\quad u_1[i_{s1}] = u_1 \wedge \ldots \wedge u_m[i_{s1}] = u_m$$
$$\quad \textbf{then } \overline{c_s[i_s]}\langle y[u_{s1}]\rangle$$
$$\quad \textbf{else new } y : T; \overline{c_s[i_s]}\langle y\rangle$$

where the channel c_s and the variables $u_1, \ldots, u_m, u_{s1}, y$ do not occur in Q and the variable x has type T and is defined under replications $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$ in Q.

The process Q preserves the secrecy of x with public variables V ($x \notin V$) if and only if $Q \mid R_x^0 \approx^V Q \mid R_x^1$.

This lemma provides a characterization of the secrecy of a variable x, defined under replications $!^{i_1 \leq n_1} \ldots !^{i_m \leq n_m}$, so x is actually an array with indices in $[1, n_1] \times \cdots \times [1, n_m]$. The processes R_x^0 and R_x^1 allow the adversary to query the variable x: if the adversary sends indices $u_1, \ldots, u_m$ on channel $c_s[i_s]$, and $x[u_1, \ldots, u_m]$ is defined, then the process R_x^0 replies with the value of $x[u_1, \ldots, u_m]$; instead, the process R_x^1 replies with a random value. The **find** in R_x^1 makes sure that, if the indices $u_1, \ldots, u_m$ have already been queried, the previous reply is sent; otherwise, a fresh random value y is chosen in the type T of x by **new** $y : T$, and sent as a reply. The replication $!^{i_s \leq n_s}$ in R_x^0 and R_x^1 allows the adversary to perform at most n_s such queries; n_s is chosen large enough so that it is not a limitation. Lemma 1 says that x is secret with public variables V if and only if an adversary with access to variables V cannot distinguish between $Q \mid R_x^0$ and $Q \mid R_x^1$, that is, it cannot distinguish between the real values of x and independent random values. (Previous definitions did not allow public variables; this is a recent extension of CryptoVerif.)

Correspondences. Correspondences [81] are properties of executed sequence of events, such as "if some event has been executed, then some other event has been executed". They are typically used for formalizing authentication. Given a correspondence *corr*, we define a distinguisher D such that $D(\mathcal{E}, a) = \mathsf{true}$ if and only if the sequence of events $\mathcal{E}$ satisfies the correspondence *corr*. We write this distinguisher simply *corr*, and write $\neg corr$ for its negation.

Definition 2 (Correspondence)**.** *The process Q* satisfies the correspondence *corr with public variables V if and only if, for all evaluation contexts C acceptable for Q with public variables V that do not contain events used in corr,* $\Pr[C[Q] \rightsquigarrow_\eta \neg corr]$ *is negligible.*

We refer the reader to [26] for more details on the verification of correspondences in CryptoVerif. We have the following lemma:

Lemma 2. *If $Q \approx^{V \cup \{x\}} Q'$ and Q preserves the secrecy of x with public variables V, then so does Q'.*

If $Q \approx^V Q'$ and Q satisfies a correspondence corr with public variables V, then so does Q'.

6.2 Cryptographic Assumptions

We make the following assumptions about the cryptographic algorithms supported by TLS 1.3 clients and servers.

Diffie-Hellman. We assume that the Diffie-Hellman groups used in TLS 1.3 satisfy the gap Diffie-Hellman (GDH) assumption [68]. This assumption means that given g, g^a, and g^b for random a, b, the adversary has a negligible probability to compute g^{ab}, even when the adversary

has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$.

In our proof, we require GDH rather than the weaker decisional Diffie-Hellman (DDH) assumption, in order to prove secrecy of keys on the server side as soon as the server sends its `Finished` message: at this point, if the adversary controls a certificate accepted by the client, he can send its own key share y' to the client to learn information on $g^{x'y'}$, and that would be forbidden under DDH. We also require that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$ implies $y = y'$, which holds when the considered Diffie-Hellman group is of prime order. This is true for all groups currently specified in TLS 1.3 [67, 14, 49, 47], and our proof requires it for all groups included in the future.

We also assume that all Diffie-Hellman group elements have a binary representation different from $0^{len_{\mathsf{H}()}}$. This assumption simplifies the proof by avoiding a possible confusion between handshakes with and without Diffie-Hellman exchange. Curve25519 does have a 32-byte zero element, but excluding zero Diffie-Hellman shared values is already recommended to avoid points of small order [60].

Finally, we assume that all Diffie-Hellman group elements have a binary representation different from $len_{\mathsf{H}()} \| \text{“}\texttt{TLS 1.3,}\text{”} \| l \| h \| \texttt{0x01}$. This helps ease our proofs by avoiding a collision between $\mathsf{hkdf\text{-}extract}(es, e)$ and $\mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ or $\mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$. This assumption holds with the currently specified groups and labels, since group elements have a different length than the bitstring above. The technical problem identified by our assumption was independently discovered and discussed on the TLS mailing list [73], and has led to a change in Draft-19 which makes this assumption unnecessary: in the key schedule, hkdf-extract is applied to the result of $\mathsf{derive\text{-}secret}(es, \mathsf{ds}, \text{“”})$ instead of applying it to es.

Signatures. We assume that the function sign is unforgeable under chosen-message attacks (UF-CMA) [48]. This means that an adversary with access to a signature oracle has a negligible probability of forging a signature for a message not signed by the signature oracle. Only the oracle has access to the signing key; the adversary has the public key.

Hash Functions. We assume that the function H is collision-resistant [40]: the adversary has a negligible probability of finding two different messages with the same hash.

HMAC. We need two assumptions on HMAC-H:

We require that the functions $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are independent random oracles, in order to justify the use of HMAC-H as a randomness extractor in the HKDF construct. This assumption can itself be justified as follows. Assuming that the compression function underlying the hash function is a random oracle, Theorem 4.4 in [42] shows that HMAC is indifferentiable [37] from a random oracle, provided the MAC keys are less than the block size of the hash function minus one, which is true for HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. It is then easy to show that $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are indifferentiable from independent random oracles in this case.

We assume that HMAC-H is a pseudo-random function (PRF) [10], that is, HMAC-H is indistinguishable from a random function provided its key is random and used only in HMAC-H, when the key is different from $0^{len_{\mathsf{H}()}}$ and kdf_0. We avoid these two keys to avoid confusion with the two random oracles above. Since keys are chosen randomly with uniform probability from a set key (with cardinality $|\mathsf{key}|$), the only consequence of avoiding these keys is that $\frac{2}{|\mathsf{key}|}$ is added to the probability of breaking the PRF assumption.

Authenticated Encryption. The authenticated encryption scheme is IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) [12], provided the same nonce is never used twice with the same key. IND-CPA means that the adversary

has a negligible probability of distinguishing encryptions of two distinct messages of the same length that it has chosen. INT-CTXT means that an adversary with access to encryption and decryption oracles has a negligible probability of forging a ciphertext that decrypts successfully and has not been returned by the encryption oracle.

6.3 Lemmas on Primitives and on the Key Schedule

We show the following properties:

- $\mathsf{mac}^k_{\mathsf{H}}(m) = \mathsf{mac}^k(\mathsf{H}(m))$ is an SUF-CMA (strongly unforgeable under chosen message attacks) MAC. Indeed, since $\mathsf{mac} = \mathsf{HMAC\text{-}H}$ is a PRF, it is an SUF-CMA MAC as shown in [11], and this property is preserved by composition with a collision-resistant hash function.

- $\mathsf{sign}^{sk}_{\mathsf{H}}(m) = \mathsf{sign}^{sk}(\mathsf{H}(m))$ is an UF-CMA signature. Indeed, sign is an UF-CMA signature, and this property is preserved by composition with a collision-resistant hash function.

We also prove several lemmas on the key schedule of TLS 1.3, using CryptoVerif.

- When es is a fresh random value, $e \mapsto \mathsf{hkdf\text{-}extract}(es, e)$ and $log_1 \mapsto \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$ are indistinguishable from independent random functions, and $k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ and $\mathsf{hkdf\text{-}extract}(es, 0^{len_{\mathsf{H}()}})$ are indistinguishable from independent fresh random values independent from these random functions.

- When hs is a fresh random value, $log_1 \mapsto \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1) \| \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$ is indistinguishable from a random function and $\mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$ is indistinguishable from a fresh random value independent from this random function.

- When ms is a fresh random value, $log_4 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$ and $log_7 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$ are indistinguishable from independent random functions.

- When l_1, l_2, l_3 are pairwise distinct labels and s is a fresh random value, we have that the keys $\mathsf{hkdf\text{-}expand\text{-}label}(s, l_i, \text{“”})$ for $i = 1, 2, 3$ are indistinguishable from independent fresh random values.

All random values considered above are uniformly distributed. We use these properties as assumptions in our proof of the protocol. This modular approach considerably reduces the complexity of the games that CryptoVerif has to consider.

These results suggest that the key schedule could be simplified by replacing groups of calls to derive-secret that use the same key and log with a single call to derive-secret that would output the concatenation of several keys. The same remark also holds for calls to hkdf-expand-label that use the same key. This approach corresponds to the usage of expansion recommended in the formalization of HKDF [55], and would simplify the proof: some lemmas above would no longer be needed. We would also recommend replacing $ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$ with $ms = \mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$: that would be more natural since we use the PRF property of HMAC-H for this computation and not the randomness extraction. If the argument $0^{len_{\mathsf{H}()}}$ may change in the future, then we would support the recommendation of applying hkdf-extract to the result of $\mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$, discussed on the TLS mailing list [73] and implemented in Draft-19.

6.4 Verifying 1-RTT Handshakes without Pre-Shared Keys

To prove the security of TLS 1.3 in CryptoVerif, we split the protocol into three parts, as shown in Figure 10, and verify them in sequence, before composing them by hand into a proof for the full protocol. This modular hybrid approach allows us to have proofs of manageable complexity, and to obtain results even when keys are reused many times, such as when several PSK-based resumptions are performed, which would otherwise be out of scope of CryptoVerif.

We first consider the initial 1-RTT handshake shown in Figure 4, until the new client and server session boxes. We model a honest client and a honest server, which are willing to interact with each other, but also with dishonest clients and servers included in the adversary. We do not consider details of the negotiation (or the `RetryRequest` message). We give the handshake keys (k_c^h and k_s^h) to the adversary, and let it encrypt and decrypt the handshake messages, so our security proof does not rely on the encryption of the handshake.

We assume that the server is always authenticated and consider both the handshake with and without client authentication. The honest client and server may be compromised at any time: the secret key of the compromised participant is then sent to the adversary, and the compromise is recorded by defining a variable `corruptedClient` or `corruptedServer`.

The outputs of this protocol are the application traffic secrets ats_c and ats_s (the derivation of the keys k_c and k_s from these secrets is left for the record protocol), the exporter master secret ems, and the resumption master secret psk' (later used as pre-shared key).

This protocol is modeled in CryptoVerif as shown in Figure 8. The context C_h first chooses randomly a key hk that models the choice of the random oracle HKDF_extract_zero_salt, that is, $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$. Then, it provides the process Q_h, which allows the adversary to query this random oracle by sending its query on channel $c_{h3}[i_h]$ and receiving the result on channel $c_{h4}[i_h]$. This random oracle is actually not used in the initial handshake; adding it simplifies the composition with the handshake with pre-shared keys which uses it. The context C generates keys, defines the random oracle $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ in the same style as C_h, deals with key compromise, and runs client and server processes. We omit the details of this context to focus on the way we specify security properties.

At the end of the client code, the context C runs the process $P_{ClientFinal}$. This process executes event ClientTerm1 with three arguments: the messages until the server `Finished` message (log_4), the messages after the server `Finished` message until the client `Finished` message (messages in log_7 but not in log_4), and the session keys. This event means that the client terminates. It is used for the unique channel identifier property. Next, the process distinguishes two cases, depending on whether the client is in a honest session or not. We say that the *client is in a honest session* when the certificate it received is the one of the honest server, and either this server is not corrupted or the messages received by the client come from the honest server. Intuitively, the client is in a honest session when it talks to the honest server. In this case, the client executes event ClientTerm with arguments the messages until the server `Finished` message (log_4) and the session keys (psk' excluded). It also executes event ClientAccept with arguments all messages (log_7) and all session keys. These events are used for key authentication. It stores the keys ats_c (that is, $cats$), ems, psk' (that is, $resumption\_secret$) in variables c_cats, c_ems, $c\_resumption\_secret$ respectively. We shall prove the secrecy of these variables. Finally, it outputs the final message (client `Certificate`, `CertVerify`, `Finished`). When the client is not in a honest session, it just outputs the final message and the keys ats_c, ats_s, ems, psk', so that the adversary can continue the protocol. No security property is proved in this case.

On the server side, the context C runs the server until it is ready to send the server `Finished` message and 0.5-RTT data. Then it executes the process $P_{Server0.5-RTT}$. This process distinguishes two cases, depending on whether the Diffie-Hellman key share $g^{x'}$ received by the server

$Q_{IH} = C_h[C[P_{ClientFinal}, P_{Server0.5-RTT}]]$

$C_h = c_{h1}(); \mathbf{new}\ hk : T_{hk}; \overline{c_{h2}}\langle\rangle; ([\,] \mid Q_h)$
$Q_h = !^{i_h \leq n_h} c_{h3}[i_h](x : T_h); \overline{c_{h4}[i_h]}\langle \mathsf{HKDF\_extract\_zero\_salt}(hk, x)\rangle$

$P_{ClientFinal} =$
  event ClientTerm1$((cr, cgx, sr, cgy, log0, log1, scv, m), final_log, (client_hk,$
    $server\_hk, client\_hiv, server\_hiv, cfk, sfk, cats, c\_sats, ems, resumption\_secret))$;
  if *honestsession* **then**
  (
1: **event** ClientTerm$((cr, cgx, sr, cgy, log0, log1, scv, m), (client_hk,$
      $server\_hk, client\_hiv, server\_hiv, cfk, sfk, cats, c\_sats, ems))$;
2: **event** ClientAccept$((cr, cgx, sr, cgy, log0, log1, scv, m, final_log), (client_hk,$
      $server\_hk, client\_hiv, server\_hiv, cfk, sfk, cats, c\_sats, ems, resumption\_secret))$;
    let $c\_cats : key = cats$ **in**
    let $c\_ems : key = ems$ **in**
    let $c\_resumption\_secret : key = resumption\_secret$ **in**
    $\overline{io6[i_C]}\langle final\_mess\rangle$
)
  else
3: $\overline{io7[i_C]}\langle(final\_mess, (resumption\_secret, cats, c\_sats, ems))\rangle$.

$P_{Server0.5-RTT} =$
  find $j \leq N_C$ **suchthat defined**$(cgx[j]) \wedge sgx = cgx[j]$ **then**
  (
4: **event** ServerAccept$((cr, sgx, sr, sgy, log0, log1, scv, m), (client_hk,$
      $server\_hk, client\_hiv, server\_hiv, cfk, sfk, cats, sats, ems))$;
    let $s\_sats : key = sats$ **in**
    $\overline{io25[i_S]}\langle(\mathsf{ServerCertificateVerifyOut}(scv), \mathsf{ServerFinishedOut}(m))\rangle$;
    $Q_{ServerAfter0.5RTT1}$
)
  else
5: $\overline{io25'[i_S]}\langle((\mathsf{ServerCertificateVerifyOut}(scv), \mathsf{ServerFinishedOut}(m)), sats)\rangle$;
    $Q_{ServerAfter0.5RTT2}$

$Q_{ServerAfter0.5RTT1} = C'[$
  event ServerTerm1$((cr, sgx, sr, sgy, log0, certS, log1, scv, m), clientfinished, (client_hk,$
    $server\_hk, client\_hiv, server\_hiv, cfk, sfk, s\_cats1, sats, s\_ems1, s\_resumption\_secret1))$;
  if *honestsession* **then**
6: **event** ServerTerm$((cr, sgx, sr, sgy, log0, certS, log1, scv, m, clientfinished), (client_hk,$
      $server\_hk, client\_hiv, server\_hiv, cfk, sfk, s\_cats1, sats, s\_ems1, s\_resumption\_secret1))$;
    $\overline{io27[i_S]}\langle\rangle$
  else
7: $\overline{io28[i_S]}\langle(s\_resumption\_secret1, s\_cats1, sats, s\_ems1)\rangle]$

Figure 8: Model of the initial 1-RTT handshake

comes from the honest client. If it does, it executes the event ServerAccept with arguments the messages until the server `Finished` message (log_4) and the session keys (psk' excluded). It stores the key ats_s (that is, $sats$) in s_sats. We shall prove secrecy of this key. Finally, it outputs the server `CertVerify` and `Finished` messages. When the Diffie-Hellman key share received by the server does not come from the honest client, the server outputs the `CertVerify` and `Finished` messages and the key ats_s so that the adversary can send 0.5-RTT data by himself. We do not prove security of 0.5-RTT data in this case.

After that, the server continues with $Q_{ServerAfter0.5RTT1}$ or $Q_{ServerAfter0.5RTT2}$. The process $Q_{ServerAfter0.5RTT1}$ executes event ServerTerm1 with three arguments: the messages until the server `Finished` message (log_4), the messages after the server `Finished` message until the client `Finished` message (messages in log_7 but not in log_4), and the session keys. Next, the process distinguishes two cases, depending on whether the server is in a honest session or not. We say that the *server is in a honest session* when

- if the client is authenticated, the certificate received by the server is the one of the honest client, and either this client is not corrupted or the messages received by the server come from the honest client;
- if the client is not authenticated, the Diffie-Hellman key share $g^{x'}$ received by the server comes from the honest client.

Intuitively, the server is in a honest session when it talks to the honest client. In this case, the server executes event ServerTerm with arguments all messages (log_7) and all session keys, and finally outputs an empty message to return control to the adversary. When the server is not in a honest session, it outputs the keys (psk', ats_c, ats_s, ems), so that the adversary can continue the protocol. No security property is proved in this case.

The process $Q_{ServerAfter0.5RTT2}$ is similar to $Q_{ServerAfter0.5RTT1}$, but with renamed channels (so that all channels are distinct) and variables $s\_resumption\_secret1, s\_cats1, s\_ems1$ renamed into $s\_resumption\_secret2, s\_cats2, s\_ems2$. This renaming is necessary because, when we prove secrecy of a variable, CryptoVerif requires that it is defined at a single location of a game.

Let us define $V_{\text{in}} = \{c\_cats,\ c\_sats,\ c\_ems,\ c\_resumption\_secret,\ s\_cats1,\ s\_cats2,\ s\_sats,\ s\_ems1,\ s\_ems2,\ s\_resumption\_secret1,\ s\_resumption\_secret2\}$ the set of variables containing the keys ats_c, ats_s, ems, psk' in honest sessions, on the client and server sides.

CryptoVerif proves the following properties:

- **Key Authentication:** If the client terminates and is in a honest session, then the server has accepted a session with the honest client, and they share the same parameters: the keys ats_c, ats_s, and ems and all messages sent in the protocol until the server `Finished` message. (We can make no claim on the client `Finished` message because it has not been received by the server at this point, nor on psk' because it depends on the client `Finished` message.) Formally, CryptoVerif proves the correspondence

$$\textbf{inj-event}(\mathsf{ClientTerm}(log_4, keys)) \Longrightarrow \textbf{inj-event}(\mathsf{ServerAccept}(log_4, keys)) \qquad (1)$$

  with public variables V_{in}, which means that, with overwhelming probability, each execution of event ClientTerm corresponds to a distinct execution of event ServerAccept with the same arguments. Event ClientTerm is executed when the client terminates and is in a honest session, event ServerAccept is executed when the server accepts a session with the client, and their arguments are the messages until server `Finished` and keys including ats_c, ats_s, and ems.

Conversely, if a server terminates and is in a honest session, then the client has accepted a session with the honest server, and they agree on the established keys and on all messages sent in the protocol. Formally, CryptoVerif proves the correspondence

$$\textbf{inj-event}(\mathsf{ServerTerm}(log_7, keys)) \Longrightarrow \textbf{inj-event}(\mathsf{ClientAccept}(log_7, keys)) \tag{2}$$

with public variables V_{in}, which means that, with overwhelming probability, each execution of event ServerTerm corresponds to a distinct execution of event ClientAccept with the same arguments.

- **Replay Prevention:** The authentication properties stated above are already injective (because of the presence of **inj-event**), that is, they guarantee that each session of the client (resp. server) corresponds to a distinct session of the server (resp. client), and consequently, they forbid replay attacks.

- **(Forward) Secrecy of Keys:** The keys ats_c, ats_s, ems, and psk' exchanged in several protocol sessions are indistinguishable from independent fresh random values. This property means for instance that the keys ats_s remains secret (indistinguishable from independent fresh random values) even if ats_c, ems, psk' are given to the adversary, and similarly for the other keys.

  We prove secrecy of ats_s on the server side when the key share $g^{x'}$ comes from the client as soon as the server sends its `Finished` message. This property allows us to prove security of 0.5-RTT messages by composition with the record protocol. Secrecy holds on the client side as well, when the client is in a honest session, because the client uses the same key as the server by key authentication. Formally, CryptoVerif proves that the protocol preserves the secrecy of s_sats with public variables $V_{\text{in}} \setminus \{c\_sats, s\_sats\}$.

  As noted in Section 3, the authentication for 0.5-RTT messages is weak: the client is not authenticated yet, so in the proof of secrecy of ats_s, we require that the key share $g^{x'}$ comes from the client. That weakens the authentication guarantees for all data received by an authenticated client. We have also written an alternative model without 0.5-RTT messages, in which we prove secrecy of ats_s on the client side when the client is in a honest session.

  Similarly, we prove secrecy of ats_c, ems, and psk' on the client side when the client is in a honest session. Secrecy holds on the server side as well, when the server is in a honest session, because the server uses the same key as the client by key authentication. Formally, CryptoVerif proves that the protocol preserves the secrecy of c_cats with public variables $V_{\text{in}} \setminus \{c\_cats, s\_cats1, s\_cats2\}$, of c_ems with public variables $V_{\text{in}} \setminus \{c\_ems, s\_ems1, s\_ems2\}$, and of $c\_resumption\_secret$ with public variables $V_{\text{in}} \setminus \{c\_resumption\_secret, s\_resumption\_secret1, s\_resumption\_secret2\}$.

- **Same Keys:** If the client terminates and is in a honest session and the server accepts a session with the honest client with the same message until the server `Finished` message, then the client and server have the same keys ats_c, ats_s, and ems. Formally, CryptoVerif proves the correspondence

$$\textbf{event}(\mathsf{ClientTerm}(log_4, c\_keys)) \wedge \textbf{event}(\mathsf{ServerAccept}(log_4, s\_keys)) \Longrightarrow c\_keys = s\_keys \tag{3}$$

  with public variables V_{in}.

  If the server terminates and is in a honest session and the client accepts a session with the honest client with the same message until the client `Finished` message, then the client and server

have the same keys ats_c, ats_s, ems, and psk'. Formally, CryptoVerif proves the correspondence

$$\mathbf{event}(\mathsf{ServerTerm}(log_7, s\_keys)) \wedge \mathbf{event}(\mathsf{ClientAccept}(log_7, c\_keys)) \Longrightarrow s\_keys = c\_keys \tag{4}$$

with public variables V_{in}.

The three properties key authentication with replay prevention, secrecy of keys, and same keys are standard security properties for a key exchange protocol [26].

- **Unique Channel Identifier:** When cid is psk' or $\mathsf{H}(log_7)$, we do not use CryptoVerif as the result is immediate: if a client session and a server session have the same cid, then these sessions have the same log_7 by collision-resistance of H (which implies collision-resistance of $\mathsf{HMAC\text{-}H}$), so all their parameters are equal.

  When cid is ems, collision-resistance just yields that the client and server sessions have the same log_4. CryptoVerif proves that, if a client session and a server session both terminate successfully with the same log_4, then they have the same log_7 and the same keys, so all their parameters are equal. Formally, CryptoVerif proves the correspondence

$$\begin{aligned}&\mathbf{event}(\mathsf{ClientTerm1}(sfl, c\_cfl, c\_keys)) \wedge \mathbf{event}(\mathsf{ServerTerm1}(sfl, s\_cfl, s\_keys))\\ &\quad\Longrightarrow c\_cfl = s\_cfl \wedge c\_keys = s\_keys\end{aligned} \tag{5}$$

We need to guide CryptoVerif in order to prove these properties, with the following main steps. We first apply the security of the signature under the server key sk_S. We introduce tests to distinguish cases, depending on whether the Diffie-Hellman share received by the server is a share $g^{x'}$ from the client, and whether the Diffie-Hellman share received by the client is the share g^y generated by the server upon receipt of $g^{x'}$. Then we apply the random oracle assumption on $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$, replace variables that contain $g^{x'y}$ with their values to make equality tests $m = g^{x'y}$ appear, and apply the gap Diffie-Hellman assumption. At this point, the handshake secret hs is a fresh random value. We use the properties on the key schedule established in Section 6.3 to show that the other keys are fresh random values, and apply the security of the MAC and of the signature under the client key sk_C.

6.5 Verifying Handshakes with Pre-Shared Keys

We now analyze the handshake protocol in Figure 7, up until the new client and server sessions are established. The protocol begins with 0-RTT and continues on to 1-RTT. We consider both variants of PSK-based 1-RTT, with and without Diffie-Hellman exchange.

We ignore the ticket $\mathsf{enc}^{k_t}(psk)$ and consider a honest client and a honest server that initially share the pre-shared key psk. Dishonest clients and servers may be included in the adversary. As in the previous section, we give the handshake keys (k_c^h and k_s^h) to the adversary and ignore handshake encryption. Certificates for the client and server are optional, since they are already authenticated via the psk; we do not rely on authentication in our proofs and consider that the adversary performs the signature and verification operations on certificates if they occur.

The outputs of this protocol are the client early traffic secret ets_c (the derivation of the key k_c from ets_c is left for the record protocol), the application traffic secrets ats_c and ats_s, the exporter master secret ems, and the resumption master secret psk'.

This protocol is modeled in CryptoVerif as shown in Figure 9, after moving some random number generations and assignments. The process first defines the random oracle $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$ using the context C_h as in Figure 8. Then it chooses a fresh pre-shared key

$Q_{PSKH} = C_h[io22(); \textbf{new } psk : key; \overline{io23}\langle\rangle; (C_{PSKClient}[Q_{PSKOnlyClient} \mid Q_{PSKDHEClient}] \mid C_{PSKServer}[Q_{PSKOnlyServer} \mid Q_{PSKDHEServer}])]$

$Q_{PSKOnlyClient} = C_{PSKOnlyClient}[$

1: **event** ClientEarlyAccept1$((c_cr, c_log1, c_binder, c_log1'), cets)$;
let $c\_cets : key = cets$ **in**
$\overline{io2[i_C]}\langle$ClientHelloOut$(c_cr, c_binder)\rangle$;
$C'_{PSKOnlyClient}[$
event ClientTerm1$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m), (log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems, resumption_secret))$;
let $c\_sats : key = sats$ **in**
2: **event** ClientTerm$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, c_sats, ems))$;
3: **event** ClientAccept$((c_cr, c_log1, c_binder, c_log1', sr, log2, log3, m, log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, c_sats, ems, resumption_secret))$;
let $c\_cats : key = cats$ **in**
let $c\_ems : key = ems$ **in**
let $c\_resumption\_secret : key = resumption\_secret$ **in**
$\overline{io6[i_C]}\langle$ClientFinishedOut$(cfin)\rangle]]$

$Q_{PSKOnlyServer} = C_{PSKOnlyServer}[$
$io11'[i_S]()$;
find $ucl \leq N_C$ **suchthat defined**$(c\_cr[ucl], c\_log1[ucl], c\_binder[ucl], c\_log1'[ucl]) \wedge c\_cr[ucl] = s\_cr \wedge c\_log1[ucl] = s\_log1 \wedge c\_binder[ucl] = s\_binder \wedge c\_log1'[ucl] = s\_log1'$
then
let $s\_cets2 : key =$ get_client_ets$(cets_eems)$ **in**
4: **event** ServerEarlyTerm1$((s_cr, s_log1, s_binder, s_log1'), s_cets2)$;
$\overline{io12'[i_S]}\langle\rangle$
else
let $s\_cets3 : key =$ get_client_ets$(cets_eems)$ **in**
5: **event** ServerEarlyTerm2$((s_cr, s_log1, s_binder, s_log1'), s_cets3)$;
find $us \leq N_S$ **suchthat defined**$(s\_cr[us], s\_log1[us], s\_binder[us], s\_log1'[us], s\_cets1[us]) \wedge s\_cr[us] = s\_cr \wedge s\_log1[us] = s\_log1 \wedge s\_binder[us] = s\_binder \wedge s\_log1'[us] = s\_log1'$
then
$\overline{io13'[i_S]}\langle\rangle$
else
let $s\_cets1 : key = s\_cets3$ **in**
$\overline{io14'[i_S]}\langle\rangle$,
6: **event** ServerAccept$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems))$;
let $s\_sats : key = sats$ **in**
$\overline{io18[i_S]}\langle$ServerFinishedOut$(m)\rangle$;
$C'_{PSKOnlyServer}[$
event ServerTerm1$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m), (log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, cats, sats, ems, resumption_secret))$;
let $s\_cats : key = cats$ **in**
let $s\_ems : key = ems$ **in**
let $s\_resumption\_secret : key = resumption\_secret$ **in**
7: **event** ServerTerm$((s_cr, s_log1, s_binder, s_log1', sr, log2, log3, m, log4, cfin), (client_hk, server_hk, client_hiv, server_hiv, cfk, sfk, s_cats, sats, s_ems, s_resumption_secret))$;
$\overline{io30[i_S]}\langle\rangle]]$

Figure 9: Model of the handshakes with pre-shared key

psk. (The input and output just allow the adversary to schedule this choice.) Finally, it launches processes for the client and the server, for the handshakes with and without Diffie-Hellman exchange.

The process $Q_{PSKOnlyClient}$ represents the client for a handshake with pre-shared key and without Diffie-Hellman exchange. The context $C_{PSKOnlyClient}$ builds the `ClientHello` message and computes the early traffic secret ets_c. Then the process executes the event ClientEarlyAccept1 with arguments the `ClientHello` message and ets_c (that is, *cets*), and stores *cets* in *c_cets*. These operations are useful to establish security of 0-RTT data. Then it outputs the `ClientHello` message and continues the protocol until the client `Finished` message. It executes the event ClientTerm1 with three arguments: the messages until the server `Finished` message, the messages after the server `Finished` message until the client `Finished` message, and the session keys. This event means that the client terminates. It is used for the unique channel identifier property. Next, the process stores ats_s into *c_sats*. It executes event ClientTerm with arguments the messages until the server `Finished` message and the session keys (psk' excluded). It also executes event ClientAccept with arguments all messages (log_7) and all session keys. These events are used for key authentication. It stores the keys ats_c (that is, *cats*), *ems*, psk' (that is, *resumption_secret*) in variables *c_cats*, *c_ems*, *c_resumption_secret* respectively. We shall prove the secrecy of these variables. Finally, it outputs the final message (client `Certificate`, `CertVerify`, `Finished`).

The process $Q_{PSKOnlyServer}$ represent the server for a handshake with pre-shared key and without Diffie-Hellman exchange. The context $C_{PSKOnlyServer}$ receives the `ClientHello` message, and runs two holes in parallel. The first hole deals with the reception of 0-RTT data. It distinguishes several cases. When the received `ClientHello` comes unaltered from the honest client, it stores the early traffic secret in *s_cets2*, executes event ServerEarlyTerm1 with arguments the `ClientHello` message and *s_cets2*, and returns control to the adversary. In this case, we are going to show that 0-RTT data is secure: it is authenticated and confidential, but may be replayed. Otherwise, it stores the early traffic secret in *s_cets3*, executes event ServerEarlyTerm2 with arguments the `ClientHello` message and *s_cets2*, and further distinguishes two cases: either the received `ClientHello` message has already been received before, and we are going to reuse the previous early traffic secret, or it is a new `ClientHello` message and we store the early traffic secret in *s_cets1*. We shall prove that this variable is secret, and that, when the server receives an altered `ClientHello` message, it cannot receive 0-RTT data. The second hole deals with the rest of the protocol. It first executes event ServerAccept with arguments the messages until the server `Finished` message and the session keys (psk' excluded). It stores the key ats_s in *s_sats*. We shall prove secrecy of this key, used for 0.5-RTT data. Finally, it outputs the server `CertVerify` and `Finished` messages. Then the server receives the client `Finished` message, executes event ServerTerm1 with three arguments: the messages until the server `Finished` message, the messages after the server `Finished` message until the client `Finished` message, and the session keys. It stores the keys ats_c, *ems*, psk' (that is, *resumption_secret*) in variables *s_cats*, *s_ems*, *s_resumption_secret* respectively. It executes event ServerTerm with arguments all messages and all session keys, and finally outputs an empty message.

The processes for the handshake with Diffie-Hellman exchange are similar. Obviously, they additionally perform the Diffie-Hellman exchange. The events ClientEarlyAccept1, ClientTerm, ClientAccept, ServerEarlyTerm1, ServerEarlyTerm2, ServerAccept, and ServerTerm have an additional suffix DHE. The variables have prefix *cdhe_* instead of *c_* and *sdhe_* instead of *s_*.

The set of variables V_{psk} contains all variables with prefixes *c_*, *s_*, *cdhe_*, and *sdhe_*.

We run CryptoVerif on our model to obtain the following verification results:

- **Key Authentication, Replay Prevention, Secrecy of Keys, Same Keys:** CryptoVerif shows the same properties as for the handshake without pre-shared key, with similar queries. The differences are as follows: the key *psk* is never compromised, so the client and server

are always in a honest session; the queries are duplicated for the handshake with and without Diffie-Hellman exchange; the variables are not duplicated for the distinction whether the Diffie-Hellman key share received by the server comes from the client or not in 0.5-RTT, but are duplicated for the handshake with and without Diffie-Hellman exchange.

- **0-RTT data:** CryptoVerif cannot prove authentication of ets_c. While the binder $\mathsf{mac}^{k_b}(\cdot)$ authenticates most of the client `ClientHello` message, the client may offer several pre-shared keys and send a binder for each of these keys. Only the binder for the pre-shared key selected by the server is checked. Hence the adversary may alter another of the proposed binders, yielding a different log_1 and a different ets_c on the server side. This is not a serious attack, as the record protocol will fail if ets_c does not match on the client and server sides.

  CryptoVerif cannot prove replay protection for the 0-RTT session key ets_c, and indeed the client `ClientHello` message can be replayed, yielding the same key ets_c for several sessions of the server even though there is a single session of the client.

  Secrecy of ets_c holds on the client side; on the server side, each key ets_c is indistinguishable from random, but the keys ets_c are not independent of each other since an adversary may force the server to accept several times the same key ets_c by replaying the client `ClientHello` message.

  CryptoVerif shows that, if the server receives the `ClientHello` message unaltered, then the client sent it (obviously!) and the client and server share the same early traffic secret ets_c. Formally,

  $$\mathbf{event}(\mathsf{ServerEarlyTerm1}(log1, cets)) \Longrightarrow \mathbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets)) \tag{6}$$

  with public variables V_{psk}. It shows that, if the client sends a `ClientHello` message and the server receives that `ClientHello` message then the client and server share the same early traffic secret ets_c. Formally,

  $$\mathbf{event}(\mathsf{ClientEarlyAccept1}(log1, cets)) \wedge \mathbf{event}(\mathsf{ServerEarlyTerm1}(log1, cets')) \Longrightarrow cets = cets' \tag{7}$$

  with public variables V_{psk}. It shows that ets_c is secret on the client side. Formally, it shows the secrecy of c_cets with public variables $V_{\text{psk}} \setminus \{c\_cets, s\_cets2\}$. These three properties provide security for a key exchange with one-way non-injective authentication.

  CryptoVerif also shows that, if the server receives twice the same altered `ClientHello` message, then it computes the same early traffic secret ets_c. Formally,

  $$\mathbf{event}(\mathsf{ServerEarlyTerm2}(log1, cets)) \wedge \mathbf{event}(\mathsf{ServerEarlyTerm2}(log1, cets')) \Longrightarrow cets = cets' \tag{8}$$

  with public variables V_{psk}. Finally, it shows the secrecy of s_cets1 with public variables $V_{\text{psk}} \setminus \{s\_cets1, s\_cets3\}$. These two properties deal with the case in which the `ClientHello` message is altered. They show that in this case, ets_c is fresh secret random value when that `ClientHello` is received for the first time, and that it is the same as the previous ets_c when `ClientHello` is replayed. By composition with the record protocol, they imply that the server fails to receive 0-RTT data in this case.

  CryptoVerif shows the same properties for the handshake with Diffie-Hellman exchange.

- **Forward Secrecy:** CryptoVerif is unable to prove secrecy of the keys when *psk* is compromised after the end of the session, even assuming that hkdf-extract is a random oracle. Secrecy obviously does not hold in this case for the handshake without Diffie-Hellman exchange. We believe that it still holds for the handshake with Diffie-Hellman exchange; our failure to prove it in this case is due to the current limitations of CryptoVerif.

- **Unique Channel Identifier:** We proceed as in the handshake without pre-shared key. We additionally notice that, if a client session and a server session have the same log_7, then they have the same psk. Indeed, by collision-resistance of $\mathsf{mac} = \mathsf{HMAC\text{-}H}$, they have the same k^b, so the same es, so the same psk.

6.6 Verifying the Record Protocol

The third component of TLS 1.3 is the record protocol that encrypts and decrypts messages after the new client and server sessions have been established in Figures 4 and 7.

In our model, we assume that the client and server share a fresh random traffic secret. We generate an encryption key and an initialization vector (IV), and send and receive encrypted messages using those key and IV, and a counter that is distinct for each message. (Our model is more detailed than the symbolic presentation given in the figures as we consider the IV and the counter.) We also generate a new traffic secret as specified in the key update mechanism of TLS 1.3 Draft-18 (Section 7.2).

More formally, after moving some assignments, our model of the record protocol in CryptoVerif is of the following form

$$Rec = c_1(); \mathbf{new}\ b : bool; \mathbf{new}\ ts : key; \mathbf{let}\ ts_{upd} : key = \mathsf{HKDF\_expand\_upd\_label}(ts)\ \mathbf{in}\ \overline{c_2}\langle\rangle; \\ (Q_{send}(b) \mid Q_{recv})$$

It chooses a random bit b ($\mathsf{false} = 0$ or $\mathsf{true} = 1$) and a random traffic secret ts. It computes the updated traffic secret ts_{upd}, and then provides two processes $Q_{send}(b)$ and Q_{recv}. The process $Q_{send}(b)$ receives two clear messages msg_0 and msg_1, and a counter $count$. Provided the counter has not been used for sending a previous message and the messages msg_0 and msg_1 have the same padded length, it executes the event $\mathsf{sent}_0(count, msg_b)$ and sends the message msg_b encrypted using keys derived from the traffic secret ts. The process Q_{recv} receives an encrypted message and a counter $count$. Provided the counter has not been used for receiving a previous message, it decrypts the message using keys derived from the traffic secret ts and executes event $\mathsf{received}_0(count, msg)$ where msg is the clear message. Both the emission and reception can be executed several times.

CryptoVerif proves the following properties automatically:

- **Key Secrecy:** CryptoVerif proves that the updated traffic secret is indistinguishable from a fresh random value. Formally, CryptoVerif proves that ts_{upd} is secret with public variable b.
- **Message Secrecy:** CryptoVerif proves that, when the adversary provides two sets of plaintexts m_i and m'_i of the same padded length, it is unable to determine which of the two sets is encrypted, even when the updated traffic secret is leaked. Formally, CryptoVerif proves that b is secret with public variable ts_{upd}.
- **Message Authentication:** CryptoVerif proves that, if a message msg is decrypted by the receiver with a counter $count$, then the message msg has been encrypted and sent by an honest sender with the same counter $count$. Formally, CryptoVerif proves the correspondence

$$\mathbf{event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathbf{event}(\mathsf{sent}_0(count, msg))$$

  with public variables b, ts_{upd}.
- **Replay Prevention:** CryptoVerif shows that any sent application data may be accepted at most once by the receiver. Formally, CryptoVerif proves the correspondence

$$\mathbf{inj\text{-}event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathbf{inj\text{-}event}(\mathsf{sent}_0(count, msg))$$

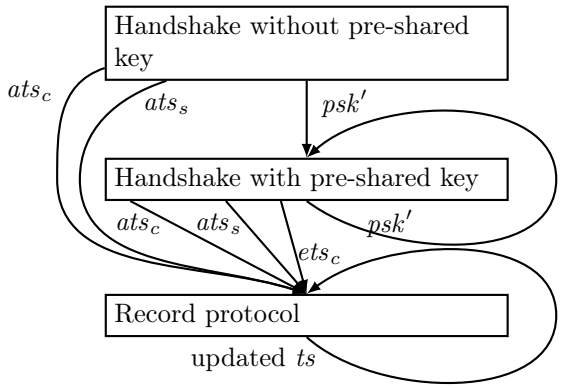


Figure 10: Structure of the CryptoVerif proof

with public variables b, ts_{upd}, which means that each execution of event $\mathsf{received}_0(count, msg)$ corresponds to a distinct execution of $\mathsf{sent}_0(count, msg)$. This correspondence implies message authentication.

We consider two other variants of the record protocol, used for 0-RTT. In the first variant, Rec_{0-RTT}, the receiver process is replicated once more, so that several sessions may have the same traffic secret, thus the receiver accepts messages with the same counter in different sessions with the same traffic secret. It models that the server may receive several times the same `ClientHello` message, yielding the same traffic secret. In this model, CryptoVerif proves key and message secrecy and message authentication but not replay prevention. In the second variant, $Rec_{0-RTT,Bad}$, the sender process is additionally removed. This model corresponds to the situation in which the `ClientHello` message is altered, and thus the server obtains a traffic secret that is not used by any client. In this model, CryptoVerif proves key secrecy, and that the $\mathsf{received}_0$ event has a negligible probability of being executed: $\mathbf{event}(\mathsf{received}_0(count, msg)) \Longrightarrow \mathsf{false}$ with public variable ts_{upd}.

6.7 A Composite Proof for TLS 1.3 Draft-18

We compose these results using a hybrid argument (as in [44]). Figure 10 summarizes the structure of the composition. We provide only a brief sketch of the composition. The detailed proof is in progress.

First, we compose the record protocol *Rec* with itself recursively, using the secrecy of the updated traffic secret, to show that the security properties of the record protocol are preserved by key updates. We also perform similar compositions for the two variants Rec_{0-RTT} and $Rec_{0-RTT,Bad}$ of the record protocol for 0-RTT. We obtain processes Rec^m, ${Rec_{0-RTT}}^m$, and ${Rec_{0-RTT,Bad}}^m$ that perform at most m key updates, for any m.

Second, we compose the handshakes with pre-shared key with the record protocol:

- using the secret key c_cats as traffic secret in the process Rec^m for 1-RTT client-to-server messages: the sender side is plugged after event ClientAccept at line 3: and the receiver side is plugged after event ServerTerm at line 7: in Figure 9;
- using the secret key s_sats as traffic secret in the process Rec^m for 0.5-RTT and 1-RTT server-to-client messages: the sender side is plugged after event ServerAccept at line 6: and the receiver side is plugged after event ClientTerm at line 2: in Figure 9;

- using the secret key *c_cets* as traffic secret in the process $Rec_{0-RTT}{}^{m}$ for 0-RTT messages when the `ClientHello` message has not been altered: the sender side is plugged after event ClientEarlyAccept1 at line 1: and the receiver side is plugged after event ServerEarlyTerm1 at line 4: in Figure 9;
- using the secret key *s_cets3* as traffic secret in the process $Rec_{0-RTT,Bad}{}^{m}$ for 0-RTT when the `ClientHello` message has been altered: the receiver process is plugged after event ServerEarlyTerm2 at line 5: in Figure 9. (There is no sender process in this case.)

We perform similar compositions in the handshake with pre-shared key and Diffie-Hellman key agreement. We also compose the obtained process with itself recursively, using the resumption secret *c_resumption_secret* as pre-shared key in the next handshake: the client side is plugged after event ClientAccept at line 3: and the server side is plugged after event ServerTerm at line 7: in Figure 9. We perform a similar composition with secret *cdhe_resumption_secret*.

For these compositions, we rely the properties of key secrecy, key authentication with replay prevention, and same key, which guarantee that the handshake with pre-shared key is a secure key exchange protocol [26]. We use the secrecy property to replace session keys with independent fresh random values. We rely on authentication with replay prevention and same key to show that the same replacement is performed in matching sessions of the client and server. For 0-RTT, the authentication is weaker (we do not have replay prevention), so we need to adapt our composition theorems to this case. The composed protocol inherits security properties from the components we compose. In particular, we obtain secrecy, authentication, and replay prevention for 0.5-RTT and 1-RTT application messages in both directions. For 0-RTT messages, since the handshake does not prevent replays for this key, the composition will not prevent replays for messages sent under this key. These compositions yield processes $Q_{PSKH}^{l,m}$ that perform at most l successive handshakes with pre-shared key and m key updates.

Third, we compose the initial handshake with the record protocol

- using the secret key *c_cats* as traffic secret in the process Rec^{m} for 1-RTT client-to-server messages: the sender side is plugged after event ClientAccept at line 2: and the receiver side is plugged after event ServerTerm at line 6: in Figure 8 and in the process $Q_{ServerAfter0.5RTT2}$;
- using the secret key *s_sats* as traffic secret in the process Rec^{m} for 0.5-RTT and 1-RTT server-to-client messages: the sender side is plugged after event ServerAccept at line 4: and the receiver side is plugged after event ClientTerm at line 1: in Figure 8.

We also compose the initial handshake with the process $Q_{PSKH,s}^{l,m}$ that runs handshakes with pre-shared key, using the secret key *c_resumption_secret* as pre-shared key: the client side is plugged after event ClientAccept at line 2: and the server side is plugged after event ServerTerm at line 6: in Figure 8 and in the process $Q_{ServerAfter0.5RTT2}$. (The forward secrecy property of the initial handshake allows us to leak the keys sk_S and sk_C, so that the adversary can indeed perform the signature operations related to certificates, as we assumed in our model of handshakes with pre-shared keys.)

Finally, we compose the obtained process with a process that runs the rest of the TLS protocol without any event or security claim, at lines 3:, 5:, 7: of Figure 8 and at a line similar to 7: in the process $Q_{ServerAfter0.5RTT2}$.

These compositions allow us to infer security properties of the TLS protocol from properties of the handshakes and the record protocol. In particular, we obtain secrecy, forward secrecy (with respect to the compromise of sk_S and sk_C), authentication, and replay prevention for 0.5-RTT and 1-RTT application messages in both directions. For 0-RTT messages, we do not obtain replay prevention, but still obtain secrecy, forward secrecy (with respect to the compromise of sk_S and sk_C), and authentication.

7 RefTLS: a Reference TLS 1.3 Implementation with a Verified Protocol Core

In today's web ecosystem, TLS is used by wide variety of client and server applications to establish secure channels across the Internet. For example, Node.js servers are written in JavaScript and can accept HTTPS connections using a Node's builtin `https` module that calls OpenSSL. Popular desktop applications, such as WhatsApp messenger, are also written in JavaScript using the Electron framework (which combines Node.js with the Chromium rendering engine); they connect to servers using the same `https` module.

Our goal is to develop a high-assurance reference implementation of TLS 1.3, called RefTLS, that can be seamlessly used by Electron apps and Node.js servers. We want our implementation to be small, easy to read and analyze, and effective as an early experimental version of TLS 1.3 that real-world applications can use to help them transition to TLS 1.3, before it becomes available in mainstream libraries like OpenSSL. Crucially, we want to be able to verify the security of the core protocol code in RefTLS, and show that it avoids both protocol-level attacks as well as implementation bugs in its protocol state machine [15].

In this section, we describe RefTLS and evaluate its progress towards these goals. RefTLS has been used as a prototype implementation of TLS Draft-13 to Draft-18, interoperating with other early TLS 1.3 libraries. Its protocol core has been symbolically analyzed with ProVerif, and it has been successfully integrated into Electron applications.

Flow and ProScript. RefTLS is written in Flow [36], a typed variant of JavaScript. Static typing in Flow guarantees the absence of a large class of classic JavaScript bugs, such as reading a missing field in an object. Consequently, our code looks very much like a program in a typed functional language like OCaml or F#. We would like to verify the security of all our Flow code, but since Flow is a fully-fledged programming language, it has loops, mutable state, and many other features that are hard to automatically verify.

In earlier work, we developed a typed subset of JavaScript called ProScript [54] that was designed for writing cryptographic protocol code that could be compiled automatically to ProVerif. ProScript is also a subset of Flow and so we can reuse its ProVerif compiler to extract symbolic models from the core protocol code in RefTLS, if we write it carefully.

ProScript code is written defensively, in that it cannot, even accidentally, access external libraries or extensible JavaScript functionalities such as object instantiation, or redefinable properties such as Array.split. These restrictions are necessary in JavaScript where external functions can completely redefine the behavior of all libraries and object prototypes. The resulting style enforces syntactic scoping and strict type checking for all variables and functions, and disallows implicit coercions, object prototype access, and dynamic extensions of arrays and objects.

For ease of analysis, ProScript disallows loops, recursion, and only allows access mutable state through a well defined `table` interface. These are significant restrictions, but as we show, the resulting language is still expressive enough to write the core composite protocol code for TLS 1.0-1.3.

Implementation Structure. Figure 11 depicts the architecture of RefTLS and shows how it can be safely integrated into larger, unverified and untrusted applications. At the top, we have Node.js and Electron applications written in JavaScript. RefTLS exposes an interface to these applications that exactly matches that of the default Node.js `https` module (which uses OpenSSL), allowing these applications to transparently use RefTLS instead of OpenSSL.

The RefTLS code itself is divided into untrusted Flow code that handles network connections and implements the API, a verified protocol module, written in ProScript, and some trusted but unverified Flow code for parsing and serializing TLS messages. All this code is statically

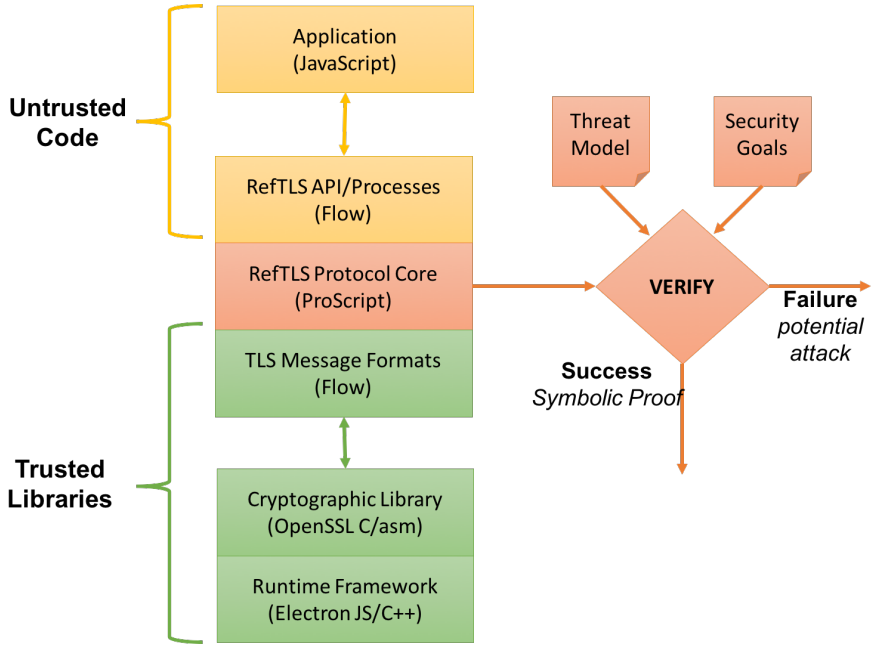


Figure 11: RefTLS Architecture. The library is written in Flow, a typed subset of JavaScript. The protocol core is verified by translation to ProVerif. The cryptographic library, message formatting and parsing, and the runtime framework are trusted. The application and parts of the RefTLS library are untrusted (assumed to be adversarial in our model).

typechecked in Flow. The core protocol module, called RefTLS-CORE, implements all the cryptographic operations of the protocol. It exposes an interface that allows RefTLS to drive the protocol, but hides all keying material and sensitive session state within the core module. This isolation is currently implemented via the Node module system; but we can also exploit Electron's multi-threading feature in order to provide thread-based isolation to the RefTLS-CORE module, allowing it to only be accessed through a pre-defined RPC interface. Strong isolation for RefTLS-CORE allows us to verify it without relying on the correctness of the rest of the RefTLS codebase.

However, RefTLS still relies on the security and correctness of the crypto library and the underlying Electron, Node.js, and JavaScript runtimes. In the future, we may be able to reduce this trusted computing base by relying on verified crypto [82], verified JavaScript interpreters [33], and least-privilege architectures, such as ESpectro [77], which can control access to dangerous libraries from JavaScript.

A Verified Protocol Core. In RefTLS-CORE, we develop, implement and verify (for the first time) a composite state machine for TLS 1.2 and 1.3 (shown in Appendix A). Each state transition is implemented by a ProScript function that processes a flight of incoming messages, changes the session state, and produces a flight of outgoing messages. For TLS 1.3 clients, these functions are `get_client_hello`, `put_server_hello`, and `put_server_finished`; servers use the functions `put_client_hello`, `get_server_finished`, and `put_client_finished`.

We then use the ProScript compiler to translate this module into a ProVerif script that looks much like the protocol models described in earlier sections of this paper. (See [54] for details of the translation.) Each pure function in ProScript translates to a ProVerif function; functions that

Benchmark	**RefTLS Client**	**Node.js Client**	**RefTLS Server**	**Node.js Server**
TLS 1.2 Handshake	16ms	7ms	8ms	6ms
TLS 1.2 Handshake	36ms		59ms	
TLS 1.3 Handshake	34ms		28ms	
Fetch 100MB	1163ms	730ms		

Figure 12: In the first benchmark, RefTLS and Node.js's HTTPS module perform handshakes against OpenSSL. In the second and third benchmarks, we test RefTLS against itself. In the fourth benchmark, the clients attempt to fetch 100MB of data from OpenSSL over TLS 1.2.

modify mutable state are translated to ProVerif processes that read and write from tables. The interface of the module is compiled to a top-level process that exposes a subset of the protocol functions to the adversary over a public channel.

The adversary can call these functions in any order and any number of times, to initiate connections in parallel, to provide incoming flights of messages, and to obtain outgoing flights of messages. The ProVerif model uses internal tables, not accessible to the attacker, to manage state updates between flights and preserve state invariants through the protocol execution.

Our approach allows us to quickly obtain verifiable ProVerif models from running RefTLS code. For example, we were able to rapidly prototype changes to the TLS 1.3 specification between Draft-13 and Draft-18, while testing for interoperability and analyzing the core protocol at the same time. In particular, we extracted a model from our Draft-18 implementation, and verified our security goals from §3 and §5 with ProVerif.

We engineered the ProScript compiler to generate readable ProVerif models that can be modified by a protocol analyst to experiment with different threat models. We are working towards applying the same automated translation approach towards CryptoVerif models. CryptoVerif syntax differs slightly from the ProVerif syntax, yet there is ongoing work in the CryptoVerif team to have it accept the same source syntax as ProVerif. However, the kind of models that are easy to verify using CryptoVerif differ from the models that ProVerif can automatically verify, and the assumptions on cryptographic primitives will always remain different. Therefore, even if the source syntax is the same, we may need to adapt our compiler to generate different models for ProVerif and CryptoVerif.

Evaluation: Verification, Interoperability, Performance. The full RefTLS codebase consists of about 6500 lines of Flow code, including 3000 lines of trusted libraries (mostly message parsing), 2500 lines of untrusted application code, and 1000 lines of verified protocol core. From the core, we extracted an 800 line protocol model in ProVerif and composed it with our generic library from §2. Verifying this model took several hours on a powerful workstation.

RefTLS implements TLS 1.0-1.3, and interoperates with all major TLS libraries for TLS 1.0-1.2. Fewer libraries currently implement TLS 1.3, but RefTLS participated in the IETF Hackathon and achieved interoperability with other implementations of Draft-14. It now interoperates with NSS (Firefox) and BoringSSL (Chrome) for Draft-18.

By implementing Node's `https` interface, we are able to naturally integrate RefTLS within any Node or Electron application. We demonstrate the utility of this approach by integrating RefTLS into the Brave web browser, which is written in Electron. We are able to intercept all of Brave's HTTPS requests and reliably fulfill them through RefTLS.

We benchmarked RefTLS against Node.js's default OpenSSL-based HTTPS stack when run against an OpenSSL peer over TLS 1.2. Our results are shown in Figure 12. In terms of compu-

tational overhead, RefTLS is two times slower than Node's native library, which is not surprising since RefTLS is written in JavaScript, whereas OpenSSL is written in C. In exchange for speed, RefTLS offers an early implementation of TLS 1.3 and a verified protocol core. Furthermore, in many application scenarios, network latency dominates over crypto, so the performance penalty of RefTLS may not be that noticeable.

8 Discussion and Related Work

Symbolic Analysis of TLS 1.3. We symbolically analyzed a composite model of TLS 1.3 Draft-18 with optional client authentication, PSK-based resumption, and PSK-based 0-RTT, running alongside TLS 1.2 against a rich threat model, and we established a series of security goals. In summary, 1-RTT provides forward secrecy, authentication and unique channel identifiers, 0.5-RTT offers weaker authentication, and 0-RTT lacks forward secrecy and replay protection.

We discovered potential vulnerabilities in 0-RTT client authentication in earlier draft versions. These attacks were presented at the TLS Ready-Or-Not (TRON) workshop and contributed to the removal of certificate-based 0-RTT client authentication from TLS 1.3. The current design of PSK binders in Draft-18 is also partly inspired by these kinds of authentication attacks.

TLS 1.3 has been symbolically analyzed before, using the Tamarin prover [39]. ProVerif and Tamarin are both state-of-the-art protocol analyzers with different strengths. Tamarin can verify arbitrary compositions of protocols by relying on user-provided lemmas, whereas ProVerif is less expressive but offers more automation. In terms of protocol features, the Tamarin analysis covered PSK and ECDHE handshakes for 0-RTT and 1-RTT in Draft-10, but did not consider 0-RTT client certificate authentication or 0.5-RTT data. On the other hand, they do consider delayed (post-handshake) authentication, which we did not consider here.

The main qualitative improvement in our verification results over theirs is that we consider a richer threat model that allows for downgrade attacks, and that we analyze TLS 1.3 in composition with previous versions of the protocol, whereas they verify TLS 1.3 in isolation.

Our full ProVerif development consists of 1030 lines of ProVerif; including a generic library incorporating our threat model (400 lines), processes for TLS 1.2 (200 lines) and TLS 1.3 (250 lines), and security queries for TLS 1.2 (50 lines) and TLS 1.3 (180 lines). All proofs complete in about 70 minutes on a powerful workstation. In terms of manual effort, these models took about 3 person-weeks for a ProVerif expert.

Computational Proofs for TLS 1.3. We presented the first mechanically-checked cryptographic proof for TLS 1.3, developed using the CryptoVerif prover. We prove secrecy, forward secrecy with respect to the compromise of long-term keys, authentication, replay prevention (except for 0-RTT data), and existence of a unique channel identifier for TLS 1.3 draft-18. Our analysis considers PSK modes with and without DHE key exchange, with and without client authentication. It includes 0-RTT and 0.5-RTT data, as well as key updates, but not post-handshake authentication.

Unlike the ProVerif analysis, our CryptoVerif model does not consider compositions of client certificates and pre-shared keys in the same handshake. It also does not account for version or ciphersuite negotiation; instead, we assume that the client and server only support TLS 1.3 with strong cryptographic algorithms. The reason we limit the model in this way is to make the proofs more tractable, since CryptoVerif is not fully automated and requires significant input from the user. With future improvements in the tool, we may be able to remove some of these restrictions.

CryptoVerif is better suited to proofs than finding attacks. In case of attack, the proof fails, but proof failure may also come from other reasons: limitations of the tool, assumptions too weak,

inappropriate guidance from the user. Therefore, we only consider in CryptoVerif properties for which ProVerif did not find attacks. Sometimes, proof failures in CryptoVerif might lead us towards computational attacks that do not appear at the symbolic level, for instance attacks that allow an adversary to distinguish between two different data messages but do not allow it to compute these messages, or attacks that rely on the algorithms of cryptographic primitives. However, we did not find such attacks in our model of TLS 1.3. We failed to prove forward secrecy for handshakes that use both pre-shared keys and Diffie-Hellman, but this failure is due to limitations in our tool, not due to an attack. Our proofs required some unusual assumptions on public values in Diffie-Hellman groups to avoid confusions between different key exchange modes; these ambiguities are inherent in Draft-18 but have been fixed in Draft-19, making some of our assumptions unnecessary.

In comparison with previous cryptographic proofs of draft versions of TLS 1.3 [44, 58, 61], our cryptographic assumptions and proof structure is similar. The main difference in this work is that our proof is mechanized, so we can easily adapt and recheck our proofs as the protocol evolves.

Our full CryptoVerif development consists of 1895 lines, including new definitions and lemmas for the key schedule (570 lines), a model of the initial handshake (710 lines), a model of PSK-based handshakes (810 lines), and a model of the record protocol (150 lines). For different proofs, we sometimes wrote small variations of these files, and we do not count all those variations here. All proofs completed in about 12 minutes. The total verification effort took about 6 person-weeks for a CryptoVerif expert.

Verifying TLS Implementations. Specifications for protocols like TLS are primarily focused on interoperability; the RFC standard precisely defines message formats, cryptographic computations, and expected message sequences. However, it says little about what state machine these protocol implementations should use, or what APIs they should offer to their applications. This specification ambiguity is arguably the culprit for many implementation bugs [15] and protocol flaws [17] in TLS.

In the absence of a more explicit specification, we advocate the need for verified reference implementations of TLS that can provide exemplary code and design patterns on how to deploy the protocol securely. We proposed one such implementation, RefTLS, for use in JavaScript applications. The core protocol code in RefTLS implements both TLS 1.2 and 1.3 and has been verified using ProVerif. However, RefTLS is a work-in-progress and many of its trusted components remain to verified. For example, we did not verify our message parsing code or cryptographic libraries, and our verification results rely on the correctness of the unverified ProScript-to-ProVerif compiler [54].

The symbolic security guarantees of RefTLS are weaker than those of computationally-verified implementations like miTLS [23]. However, unlike miTLS, our analysis is fully automated and it can quickly find attacks. The type-based technique of miTLS requires significant user intervention and is better suited to building proofs than finding attacks.

Other Verification Approaches. In addition to ProVerif and CryptoVerif, there are many symbolic and computational analysis tools that have been used to verify cryptographic protocols like TLS. As discussed above, Tamarin [75] was used to symbolically analyze TLS 1.3 Draft-10 [39]. EasyCrypt [9] has been used to develop cryptographic proofs for various components used in TLS, including the MAC-Encode-Encrypt construction used in the record layer [6].

Our ProScript-to-ProVerif compiler is inspired by previous works on deriving ProVerif models from F# [22], Java [7], and JavaScript [18]. Such translations have been used to symbolically and computationally analyze TLS implementations [20]. An alternative to model extraction is to synthesize a verified implementation from a verified model; [34] shows how to compile CryptoVerif

models to OCaml and uses it to derive a verified SSH implementation.

The most advanced case studies for verified protocol implementations use dependent type systems, because they scale well to large codebases. Refinement types for F# have been used to prove both symbolic [21] and cryptographic security properties, with applications to TLS [23]. The F* programming language [78] has been used to verify small protocols and cryptographic libraries [82]. Similar techniques have been applied to the cryptographic verification of Java programs [59].

9 Conclusion and Future Work

TLS 1.3 is a social and technical experiment in the collaborative design of a practical protocol with regular input and review from the academic research community. It seeks to reverse the traditional pattern where security analyses are performed several years after standardization, when it may be too late to change how implementations work. This paper describes our contribution to this standardization effort.

We present verification results for symbolic models in ProVerif, computational models in CryptoVerif, and a reference implementation in JavaScript of TLS 1.3 Draft-18. There are still many features and aspects of the emerging protocol standard that remain to be analyzed. Furthermore, the formal connections between our ProVerif models, CryptoVerif proofs, and JavaScript code are not as strong as could be desired. We have focused on proof automation and readable models as a pragmatic first step, but we are working on formal proofs of correctness for our translations from Flow to ProVerif and CryptoVerif, so that we can obtain strong guarantees for our protocol source code.

References

[1] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. *IEE Proceedings Information Security*, 153(1):27–39, Mar. 2006.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 5–17, 2015.

[3] M. R. Albrecht and K. G. Paterson. Lucky microseconds: A timing attack on Amazon's S2N implementation of TLS. In *EUROCRYPT*, pages 622–643, 2016.

[4] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the security of RC4 in TLS. In *USENIX Security Symposium*, pages 305–320, 2013.

[5] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, pages 526–540, 2013.

[6] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption (FSE)*, pages 163–184, 2016.

[7] M. Avalle, A. Pironti, R. Sisto, and D. Pozza. The Java SPI framework for security protocol implementation. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 746–751, Aug 2011.

[8] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: breaking TLS using SSLv2. In *USENIX Security Symposium*, pages 689–706, 2016.

[9] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII (FOSAD)*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2014.

[10] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology (CRYPTO)*, pages 602–619, 2006.

[11] M. Bellare, J. Kilian, and P. Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, Dec. 2000.

[12] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT'00*, pages 531–545, 2000.

[13] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology (Eurocrypt)*, pages 409–426, 2006.

[14] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography (PKC)*, pages 207–228, 2006.

[15] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[16] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Z. Béguelin. Downgrade resilience in key-exchange protocols. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 506–525, 2016.

[17] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 98–113, 2014.

[18] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium*, pages 653–670, 2013.

[19] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti. Verified contributive channel bindings for compound authentication. In *Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[20] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified cryptographic implementations for TLS. *ACM TOPLAS*, 15(1):3:1–3:32, 2012.

[21] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 445–456, 2010.

[22] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1), 2008.

[23] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, 2013.

[24] K. Bhargavan and G. Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 456–467, 2016.

[25] K. Bhargavan and G. Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[26] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 97–111, 2007.

[27] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

[28] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[29] B. Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *25th IEEE Computer Security Foundations Symposium (CSF'12)*, pages 325–339, Cambridge, MA, USA, June 2012. IEEE.

[30] B. Blanchet. Security protocol verification: Symbolic and computational models. In *Principles of Security and Trust (POST)*, pages 3–29, 2012.

[31] B. Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, Oct. 2016.

[32] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998.

[33] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised javascript specification. In *ACM Symposium on the Principles of Programming Languages (POPL)*, pages 87–100, 2014.

[34] D. Cadé and B. Blanchet. Proved generation of implementations from computationally secure protocol specifications. *Journal of Computer Security*, 23(3):331–402, 2015.

[35] S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 172–185. IEEE, 2009.

[36] A. Chaudhuri. Flow: Abstract interpretation of javascript for type checking and beyond. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2016.

[37] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Advances in Cryptology (CRYPTO)*, pages 430–448, 2005.

[38] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, 2011.

[39] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 470–485, 2016.

[40] I. B. Damgård. A design principle for hash functions. In *Advances in Cryptology–CRYPTO'89*, pages 416–427, 1989.

[41] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[42] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In *Advances in Cryptology (Crypto)*, pages 348–366, 2012.

[43] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[44] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1197–1210, 2015.

[45] M. Fischlin and F. Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1193–1204, 2014.

[46] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 452–469, 2016.

[47] D. Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for Transport Layer Security (TLS), Aug. 2016. `http://tools.ietf.org/rfc/rfc7919`.

[48] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[49] M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, June 2015. Available at `https://eprint.iacr.org/2015/625`.

[50] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-based multiplexed and secure transport, 2016. IETF Internet Draft.

[51] K. E. Hickman. The SSL protocol, 1995. IETF Internet Draft, `https://tools.ietf.org/html/draft-hickman-netscape-ssl-00`.

[52] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO 2012*, pages 273–293, 2012.

[53] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1185–1196, 2015.

[54] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[55] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology (CRYPTO)*, pages 631–648. 2010.

[56] H. Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in tls 1.3). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1438–1450, 2016.

[57] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO 2013*, pages 429–448, 2013.

[58] H. Krawczyk and H. Wee. The OPTLS protocol and TLS 1.3. In *IEEE European Symposium on Security & Privacy (Euro S&P)*, 2016. Cryptology ePrint Archive, Report 2015/978.

[59] R. Küsters, T. Truderung, and J. Graf. A framework for the cryptographic verification of Java-like programs. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 198–212, 2012.

[60] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. IRTF RFC 7748 `https://tools.ietf.org/html/rfc7748`, Jan. 2016.

[61] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu. Multiple handshakes security of TLS 1.3 candidates. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 486–505, 2016.

[62] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 214–231, 2015.

[63] U. Maurer and B. Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 505–515, 2010.

[64] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, 2012.

[65] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*, pages 733–748. USENIX Association, 2014.

[66] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[67] NIST. FIPS 186-3: Digital Signature Standard (DSS). Available at `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, June 2009.

[68] T. Okamoto and D. Pointcheval. The gap-problems: a new class of problems for the security of cryptographic schemes. In *Practice and Theory in Public Key Cryptography (PKC)*, pages 104–118, 2001.

[69] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, pages 372–389, 2011.

[70] K. G. Paterson and T. van der Merwe. Reactive and proactive standardisation of TLS. In *Security Standardisation Research (SSR)*, pages 160–186, 2016.

[71] M. Ray, A. Pironti, A. Langley, K. Bhargavan, and A. Delignat-Lavaud. Transport Layer Security (TLS) session hash and extended master secret extension, 2015. IETF RFC 7627.

[72] E. Rescorla. 0-RTT and Anti-Replay. `https://www.ietf.org/mail-archive/web/tls/current/msg15594.html`, Mar. 2015.

[73] E. Rescorla. [TLS] PR#875: Additional Derive-Secret stage. `https://www.ietf.org/mail-archive/web/tls/current/msg22373.html`, Feb. 2017.

[74] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. TLS renegotiation indication extension. IETF RFC 5746, 2010.

[75] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 78–94, 2012.

[76] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. IACR Cryptology ePrint Archive, 2004. `http://eprint.iacr.org/2004/332`.

[77] D. Stefan. Espectro project description, 2016. `https://cseweb.ucsd.edu/~dstefan/#projects`.

[78] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[79] M. Vanhoef and F. Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, pages 97–112, 2015.

[80] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Electronic Commerce*, 1996.

[81] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proceedings IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 1993.

[82] J. K. Zinzindohoue, E. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 296–309, 2016.

A RefTLS Protocol State Machines

Client. The RefTLS client implements the composite state machine shown in Figure 13 for TLS 1.3 and TLS 1.2. Each state represents a point in the protocol where the client is either waiting for a flight of handshake messages from the server, or it has new session keys that it wishes to communicate to the record layer. Each arrow is annotated with the name of the function in RefTLS-CORE API that implements the corresponding state transition. Each transition may involve processing a flight of incoming messages, changing the session state, and producing a flight of outgoing messages.

Server. The RefTLS server implements a dual state machine for TLS 1.3 and TLS 1.2, as depicted in Figure 14. The server decides which protocol version and key exchange the handshake

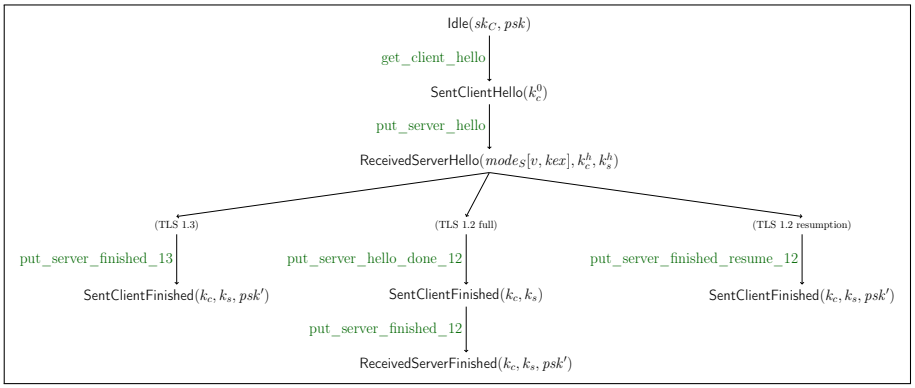


Figure 13: Client state machine

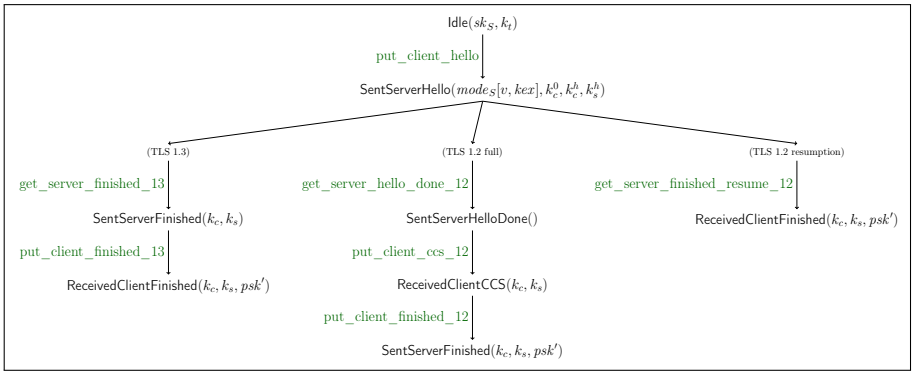
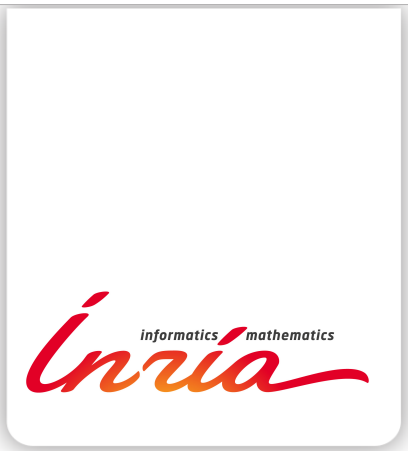


Figure 14: Server state machine

will use, and triggers the appropriate branch in the state machine by sending a `ServerHello`. Like the client, each of its state transition functions corresponds either to a flight of messages or to a change of keys.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399